Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Karol Hrdina

Vliv cache na efektivitu třídění

Katedra softwarového inženýrství

Vedoucí diplomové práce: *RNDr. Alena Koubková, Csc.*

Studijní program: *Informatika, Softwarové systémy*

Na tomto mieste by som rád poďakoval RNDr. Aleně Koubkové, Csc. za jej cenný čas a rady.

Prohlašuji, že jsem svou diplomovou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 17.4.2009 *Karol Hrdina*

Obsah

Název práce: Vliv cache na efektivitu třídění
Autor: Karol Hrdina
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Alena Koubková, Csc.
e-mail vedoucího: koubkova@ksi.ms.mff.cuni.cz

Abstrakt:

Klasické algoritmy pre triedenie vo vnútornej pamäti boli navrhnuté za predpokladu, že táto pamäť je homogénna. V moderných počítačoch je ale štruktúra pamäte hierarchická s rozdielnou rýchlosťou jednotlivých vrstiev. Doba výpočtu algoritmu teda závisí nielen na počte vykonaných operácií (napr. porovnanie prvkov), ale aj na počte presunov dát medzi jednotlivými vrstvami. Interné algoritmy tak získavajú niektoré rysy algoritmov externých.

V tejto práci si kladieme za úlohu stručne zhrnúť existujúce prístupy k problematike a opísať známe vylepšenia niektorých algoritmov pre prácu v nehomogénnej pamäti. Hlavný dôraz je kladený na implementáciu vybraných algoritmov a ich experimentálne overenie.

Kľúčové slová: cache, multimergesort, multiquicksort, triedenie.

Title: Effects of caches on performance of sorting
Author: Karol Hrdina
Department: Department of Software Engineering
Supervisor: RNDr. Alena Koubková, Csc.
Supervisor's e-mail address: koubkova@ksi.ms.mff.cuni.cz
Abstract:

Classical algorithms for sorting in internal memory were designed with an assumption, that the memory is homogenous. But modern computers have hierarchically structured memory with various speeds of it's layers. Execution time of algortihm is dependant not only on operation count, but also on count of transfers between memory layers. Therefore internal algorithms are having some characteristics of external algorithms.

In this paper we set our goal to summarize some existing approaches to this problem and summarize known optimalizations of internal sorting algorithms. Our main goal however is to impelent chosen algorithms and measure their performance experimentally.

Keywords: multiquicksort, funnelsort, sorting, cache

Kapitola 1

Úvod

Témou tejto diplomovej práce je skúmanie vplyvu cache pamätí na vnútorné triedenie. Našou hlavnou úlohou je v prvom rade prakticky overiť niektoré teoretické výsledky v tejto oblasti a ďalej zhrnúť existujúce prístupy k riešeniu problematiky.

V Kapitole 2 sú rozobrané aspekty moderných pamäťových systémov. V Kapitole 3 uvádzame prehľad výpočtových modelov. V Kapitole 4 sú uvedené dosiahnuté teoretické výsledky vo forme algoritmov. Kapitola 5 sumarizuje spôsob vykonávania a výsledky našich testov.

Kapitola 2

Pamäťové systémy a latencia

Cache-oblivious algoritmy sú analyzované v ideal-cache modeli a tento model je abstrakciou reálnych pamäťových hierarchií. Pochopenie moderných pamäťových systémov je preto nevyhnutné k pochopeniu cache-oblivious prístupu.

V Sekcii 2.1 sú popísané tie aspekty moderných pamäťových systémov, ktoré sú dôležité s ohľadom na cache-oblivious algoritmy. V Sekcii 2.2 je popísaný problém pamäťovej latencie. V Sekcii 2.3 ilustrujeme pomocou praktickej ukážky vplyv latencie pamäte na návrh algoritmov. V Sekcii 2.4 sú popísané aspekty symetrických multiprocesorov a NUMA architektúry z hľadiska cache pamätí.

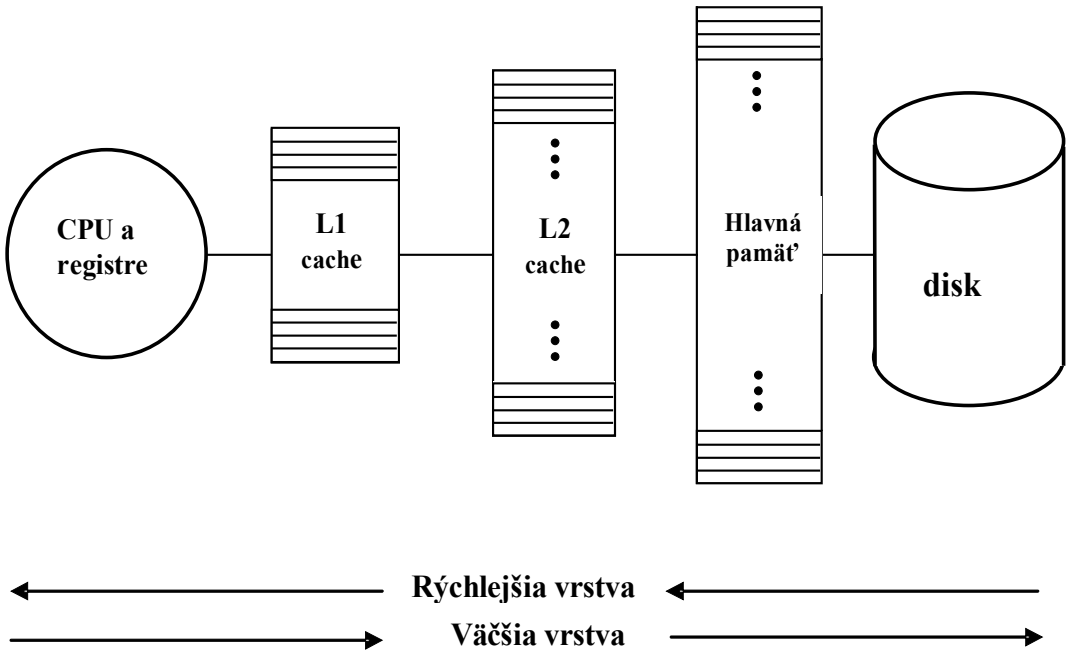
2.1 Dôležité aspekty pamäťovej hierarchie

V tejto sekcii sú popísané aspekty moderných pamäťových systémov, ktoré sú dôležité v kontexte cache-oblivious algoritmov. Dôraz sa preto nekladie na popis detailov určitého systému od nejakého výrobcu, ale skôr na popis myšlienok niektorých všeobecných a rozšírených konceptov.

Pamäťový systém obsahuje dáta, s ktorými pracuje program (vstupné, výstupné dáta) a samotný kód programu. V kontexte cache-oblivious algoritmov nás nezaujíma ako pamäťový systém ukladá programový kód. Z tohto dôvodu bol popis konceptov venujúci sa tejto téme vynechaný.

Moderné pamäťové systémy sú zložené z niekoľkých vrstiev pamätí zoradených do hierarchickej štruktúry. Typická pamäťová hierarchia je zobrazená na obrázku Obr. 2.1. Malá, ale veľmi rýchla pamäťová vrstva sa nachádza najbližšie k procesoru a jedna alebo viacero väčších, ale pomalších pamäťových vrstiev sa nachádza ďalej od procesora. Ak sa budeme dívať na registre ako na integrovanú súčasť procesora, typická pamäťová hierarchia pozostáva z jednej alebo viacerých vrstiev pamäte medzi procesorom, hlavnou pamäťou a diskom – ako najväčšou pamäťovou vrstvou. Pojem cache sa používa pri popise pamäťových vrstiev medzi procesorom a hlavnou pamäťou, takže hierarchia zobrazená na obrázku Obr. 2.1 má dve vrstvy cache a disk na celkový počet štyroch

pamäťových vrstiev. Pojem cache sa tiež používa pri popise úlohy menšej vrstvy z dvoch po sebe nasledujúcich pamäťových vrstiev, napr. hlavná pamäť slúži ako cache disku a pod. Ak nebude uvedené inak, v tejto práci pod pojmom cache rozumieme prvé z uvádzaných vysvetlení slova cache.



Obr. 2.1: Pamäťová hierarchia pozostáva z viacerých pamäťových vrstiev. Čím ďalej sa nachádza vrstva od procesora, tým je jej veľkosť väčšia a jej rýchlosť pomalšia. Pamäťová hierarchia môže mať viacej vrstiev ako je zobrazených na obrázku.

Je rozdiel medzi dočasnou časťou pamäťovej hierarchie, kde dáta nepretrvávajú po vypnutí prúdu (napr. vrstvy bližšie k procesoru) a stálou časťou pamäťovej hierarchie, kde dáta pretrvávajú nezmenené aj po vypnutí počítača (napr. disk). Dočasná časť pamäťovej hierarchie reprezentuje primárnu pamäť a disk reprezentuje sekundárnu pamäť. Niekedy sa primárna pamäť nazýva vnútorná a sekundárna pamäť vonkajšia.

V ideálnom prípade pamäťová hierarchia dodržuje vlastnosť nazývanú „*inclusion property*“. (Algoritmus dodržuje „inclusion property“, ak pre daný vstup, pamäťový stav algoritmu v každom časovom bode pre veľkosť $m+1$ zahrňuje stav m). Pre pamäťovú hierarchiu to znamená, že pamäťová vrstva bližšie k procesoru obsahuje správnu podmnožinu dát ktorejkoľvek ďalšej vrstvy. Ak si označíme vrstvy v rastúcom poradí, vrstva i sa správa ako buffer pre vrstvu $i+1$. Vrstva najďalej od procesora obsahuje všetky dáta. V tabuľke Tab. 2.2 sa pre ukážku nachádza prehľad vlastností cache pamätí niektorých procesorov rodiny Intel.

Tab. 2.2: Prehľad vlastností cache pamätí niektorých procesorov rodiny Intel. Zdroj: Rodger Dodson, Personal Reference Sheets, Lenovo - http://www.lenovoservicetraining.com

2.1.1 Princíp lokality referencií

Pamäťová hierarchia je založená na dvoch pozorovaniach ohľadne správania sa počítačových programov. Tieto pozorovania sa všeobecne nazývajú princípy lokality referencií:

temporal locality – je pozorovanie, že prístup na rovnaké pamäťové miesto je väčšinou vykonaný veľa krát v krátkom časovom úseku. Príkladom je počítadlo vo for cykle, ktoré je čítané a na ktoré je zapisované aspoň raz počas každej iterácie cyklu. Inak povedané, toto pozorovanie hovorí, že ak bolo na dané pamäťové miesto pristúpené, pravdepodobnosť, že bude na toto isté pamäťové miesto pristúpené znovu v krátkom čase je vysoká.

spatial locality – je pozorovanie, že prístup na pamäťové miesto v blízkosti pamäťového miesta, na ktoré bolo nedávno pristúpené, je veľmi pravdepodobný. Príkladom tohto pozorovania je for cyklus iterujúci cez pole prvkov.

Tieto dva princípy sú dôležité v súvislosti s cache-oblivious algoritmami, nakoľko programy, ktoré sú navrhnuté v súlade s týmito princípmi, sa správajú efektívne čo do práce s pamäťou. Akým spôsobom pamäťová hierarchia zaručuje princíp lokality, bude zrejmé z popisu kedy a ako sa dáta presúvajú medzi pamäťovými vrstvami.

2.1.2 Presúvanie a adresovanie dát v pamäťovej hierachii

Procesor môže pristupovať iba na dáta umiestnené v najbližšej pamäťovej vrstve, takže temporal locality je zachovaná udržovaním často používaných dát v čo najbližšej vrstve k procesoru.

V prípade, že si procesor vypýta dáta, ktoré sú uložené v najbližšej pamäťovej vrstve, procesor má okamžitý prístup k týmto dátam a hovoríme že nastal *cache hit*. Ak pamäťová vrstva najbližšie k procesoru neobsahuje dáta požadované procesorom, hovoríme, že nastal *cache miss* (cache výpadok). V takom prípade sa dáta musia presunúť zo vzdialenejšej pamäťovej vrstvy skôr, ako k nim bude môcť procesor pristúpiť. V zásade teda môže procesor spôsobiť cache výpadok v každej vrstve okrem poslednej. Je zrejmé, že procesor nemôže spôsobiť cache výpadok v poslednej pamäťovej vrstve, pretože posledná pamäťová vrstva obsahuje všetky dáta.

V každej pamäťovej vrstve sa obsah delí na nepretržité, po sebe idúce kusy. V cache vrstvách sa tieto kusy nazývajú *bloky* alebo *cache bloky* a môžu byť rôzne čo do veľkosti medzi jednotlivými pamäťovými cache vrstvami. Väčšinou jeden blok obsahuje niekoľko slov (pozn.: Na 32 bitovom počítači je veľkosť pamäťového miesta 32 bitov a teda je v ňom možné uložiť akúkoľvek hodnotu reprezentovateľnú 32 bitmi. V tomto zmysle je slovo rovnaké ako pamäťová bunka.).

V prípade cache výpadku sa spatial locality zachová presunutím dát medzi vrstvami po blokoch. Z toho vyplýva, že veľkosť bloku danej pamäťovej vrstvy určuje granularitu pamäťového adresovania v tej danej vrstve. Adresovanie pamäte v rámci pamäťovej hierarchie je preto iné ako adresovanie pamäti procesorom.

Čím ďalej sa pamäťová vrstva nachádza od procesora, tým je jej veľkosť väčšia a tým sú väčšie aj pamäťové „kúsky“, v ktorých sa pamäť adresuje a presúva. V hlavnej pamäťovej vrstve sú tieto kusy veľké aj niekoľko kilobytov a používa sa pre ne pojem *page* (*stránka*) namiesto bloku. Následkom toho sa neschopnosť nájsť dáta v hlavnej pamäti nazýva *page fault* (*výpadok stránky*) namiesto cache výpadku. Stránka obsahujúca požadované dáta sa potom presunie z disku do hlavnej pamäte.

Kapacita pamäťovej vrstvy je určená počtom blokov alebo stránok, ktoré môže poňať. Cache pamäte sú rozdelené na *cache lines* (*cache riadky*), ktoré môžu obsahovať každý jeden blok, takže kapacita cache vrstvy sa jednoducho spočíta vynásobením veľkosti jedného bloku s počtom cache riadkov. Hlavná pamäť je rozdelená na *frames* (*rámce*), ktoré môžu poňať každý jednu stránku. Veľkosť hlavnej pamäte je definovaná počtom stránok, ktoré môže poňať.

Ktorý blok sa má vymeniť v prípade cache výpadku určuje *associativity* (*asociativita*) a *replacement policy* (*stratégia výmeny*) pamäťovej vrstvy cache, v ktorej cache výpadok nastal.

Asociativita obmedzuje počet cache riadkov, v ktorých môže byť daný blok uložený. Cache riadky, ktoré môžu potenciálne obsahovať daný blok pamäte, nazývame *candidate lines* (*kandidátne riadky*) pre daný blok. Inými slovami, blok sa mapuje na určitý počet kandidátnych riadkov. Cache pamäte sú rozdelené do troch kategórií podľa asociativity:

fully-associative (*plne asociatívna*) cache – nekladie žiadne obmedzenia na to, kam sa môže blok umiestniť. Počet kandidátnych riadkov pre ktorýkoľvek blok je rovnaký ako celkový počet cache riadkov v cache pamäti, t.j. ktorýkoľvek blok môže byť umiestnený kamkoľvek.

direct-mapped (*priamo mapovaná*) cache – je najviac obmedzujúci typ. Každý blok má iba

jeden kandidátny riadok. Index kandidátneho riadka v cache pamäti je vypočítaný ako adresa bloku modulo počet cache riadkov.

set-associative (*množinovo-asociatívna*) cache – je hybrid plne asociatívnej cache pamäte a priamo mapovanej cache pamäte. *X*-cestná množinovo asociatívna cache pamäť je rozdelená na množiny, z ktorých každá obsahuje *x* cache riadkov. Blok pamäte sa mapuje presne na jednu z týchto množín, ale v rámci množiny môže byť blok uložený kamkoľvek, t.j. do ľubovoľného riadka. Ak je pamäť rozdelená na bloky očíslované rastúc podľa adresy od 0 do m, potom index množiny v cache pamäti, na ktorý sa mapuje blok a $(0 \leq a \leq m)$ je vypočítaný ako $a \bmod x$. Číslo x definuje stupeň asociativity cache pamäte. Typické množinovo asociatívne cache pamäte majú stupeň 2, 4, 8 príp. 16.

V prípade cache výpadku v množinovo asociatívnej alebo plne asociatívnej cache pamäti potrebujeme nejaký typ stratégie na výber cache riadka spomedzi kandidátnych riadkov (riadky pripadajúce do úvahy pre výmenu), do ktorého uložíme nový blok. Ak medzi kandidátmi existujú prázdne riadky, výber je ľahký. Ak všetky kandidátne riadky obsahujú dáta, potom sa voľba, ktorý cache riadok sa má vymeniť, robí podľa stratégie výmeny danej cache pamäte. Aby sme zachovali princíp temporal locality, optimálna stratégia výmeny by vymenila cache riadok, na ktorý je odkazované najďalej v budúcnosti. Zisťovanie, ktorý cache riadok by to mal byť, vyžaduje značnú znalosť postupnosti inštrukcií procesora v budúcnosti a preto sa v praxi využívajú jednoduchšie stratégie výmeny.

Pre 2-cestnú množinovo asociatívnu cache pamäť je implementovaná LRU (Least-Recently-Used) stratégia tak, že sa pre každý cache riadok drží jeden bit. Keď sa pristúpi na cache riadok, nastaví sa mu bit a jeho susediacim riadkom sa vynuluje. Tento bit sa nazýva „*recency bit*“. Teoreticky by mohla byť LRU stratégia implementovaná aj pre vyššie stupne asociativity, použitím viacerých „recency“ bitov. V praxi sa však používa len akási aproximácia LRU, ako napr. „not-recently-used“ stratégia. Niekedy sa dokonca používa stratégia výmeny, ktorá vyberá kandidátny riadok náhodne. Pre cache pamäte obmedzenej asociativity totiž stratégia výmeny nemá skoro žiadny dopad na počet cache výpadkov. Dokonca bolo zistené, že pre 2, 4 a 8-cestné množinovo asociatívne cache pamäte, náhodný výber riadka pre výmenu funguje skoro tak dobre ako LRU stratégia pre cache pamäte obsahujúce dáta veľkosti 16, 64 a 256 kB v 64-bytových blokoch [6]. Čím je väčšia cache pamäť, tým je menší rozdiel a pre 256 kB cache pamäte boli dve uvádzané stratégie výmeny rovnako dobré. Pre menšie veľkosti cache pamätí vyústil vyšší stupeň asociativity do menšieho počtu cache výpadkov, ale pre 256 kB cache tomu už tak nebolo.

Oveľa dôležitejšie je rozhodnúť, ktorú stránku nahradiť v hlavnej pamäti, ak dôjde k výpadku stránky. Vyhnúť sa postihu za čo i len pár výpadkov stránok v hlavnej pamäti je oveľa dôležitejšie a vyplatí sa použiť chytrú stratégiu výmeny v hlavnej pamäti. Vo vrstve hlavnej pamäte je stratégia výmeny implementovaná softwarovo a teda závisí od operačného systému.

2.1.3 Kategorizácia výpadkov cache

Výpadky cache sa delia do troch hlavných kategórií podľa toho ako nastanú:

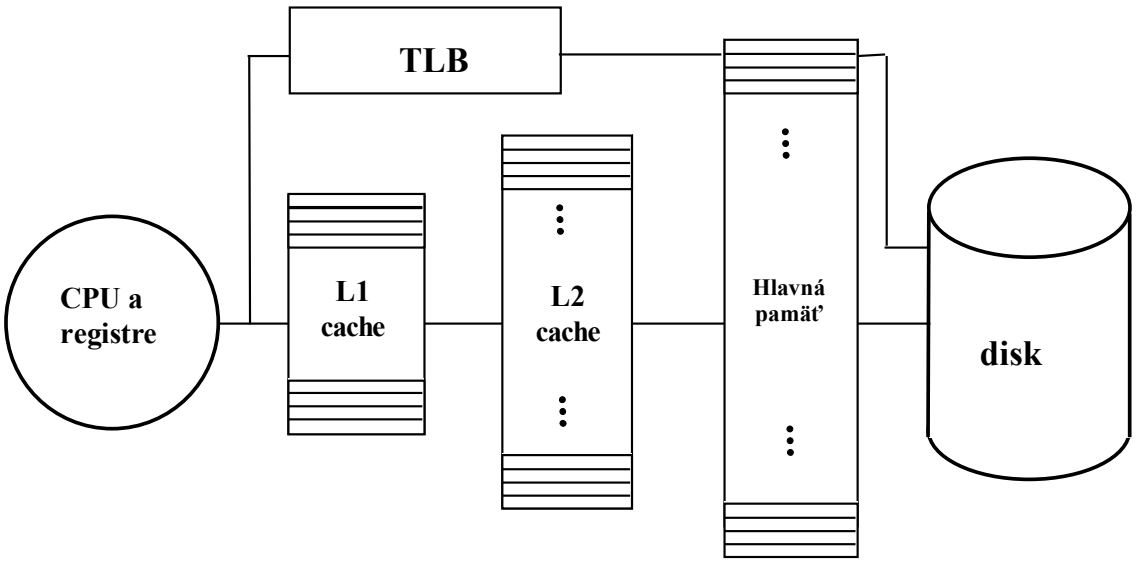
compulsory výpadky – nastávajú, ak je blok adresovaný prvý krát, tzn. nenachádzal sa predtým v cache pamäti. Tieto výpadky sú preto neodstrániteľné a dejú sa v každej vrstve pamäťovej hierarchie.

capacity výpadky – nastávajú, keď cache nemá žiadny voľný cache riadok, do ktorého by mohol byť umiestnený blok, ktorý už bol predtým v cache. Veľkosť cache pamäte a kvalita stratégie výmeny určujú počet capacity výpadkov programu. Tento typ výpadku nespôsobuje nutne výpadok v každej úrovni pamäťovej hierarchie, napr. ak bol blok vyhodený z L1 cache, stále sa ešte môže nachádzať v L2 cache.

conflict výpadky – nastávajú, ak adresovaný blok môže byť umiestnený iba do obsadeného cache riadka. Tento typ výpadkov môže nastať v cache pamätiach, ktoré nie sú plne asociatívne, aj napriek tomu, ak sa v cache nachádzajú prázdne riadky. Je to spôsobené faktom, že v množinovo asociatívnych cache pamätiach sa na jeden cache riadok mapuje viacero blokov. V plne asociatívnych cache pamätiach tento typ výpadkov nenastáva, nakoľko ktorýkoľvek blok môže byť umiestnený do ktoréhokoľvek cache riadka. Ak plne asociatívna cache pamäť nemá žiadny prázdny riadok, do ktorého by sa umiestnil blok, nastáva capacity výpadok a nie conflict výpadok.

2.1.4 Virtuálna Pamäť

Na virtuálny pamäťový systém sa môžeme pozerať ako na cachovací systém bežiaci paralelne k hierarchickému pamäťovému systému, viď obrázok Obr. 2.3. Tento paralelný systém sa stará o presuny dát medzi primárnou a sekundárnou pamäťou. Jeho úlohou je, aby si programy bežiace simultánne mysleli, že majú k dispozícií celú hlavnú pamäť. Program nevie, že sa o hlavnú pamäť delí s ostatnými programami. Ako dôsledok, systém virtuálnej pamäte odstraňuje nutnosť, aby programátor manuálne spravoval hlavnú pamäť. To znamená, že systém virtuálnej pamäte musí zabezpečiť, aby zdieľanie hlavnej pamäte prebiehalo bezpečne (napr. aby jeden program neprepísal dáta druhému programu).



Obr. 2.3: Virtuálna pamäť vytvára pamäťovú hierarchiu paralelne k hierarchii na obrázku Obr. 2.1.

Znalosť presných implementačných detailov nie je pre nás dôležitá, ale stojí za to spomenúť aspoň pár princípov:

virtuálna a fyzická adresa: Je rozdiel medzi virtuálnymi adresami, odkazujúcimi sa na imaginárny priestor, a fyzickými adresami, odkazujúcimi sa na pamäť reprezentovanú pamäťovými čipmi DRAM. Keďže každý proces by sa mal byť schopný vykonať tak, akoby mal k dispozícii celú hlavnú pamäť, každý proces má svoj vlastný virtuálny adresový priestor. Veľkosť pointeru je 32 bitov na 32 bitovej architektúre, takže počet adresovateľných pamäťových miest je obmedzený na 2^{32}, teda virtuálny adresový priestor je 4 GB. Často sa stáva, že veľa procesov beží simultánne a suma ich virtuálneho adresového priestoru je pravdepodobne väčšia ako dostupná fyzická pamäť. Systém virtuálnej pamäte sa musí stále starať o to, ktoré virtuálne stránky sa momentálne nachádzajú vo fyzickej pamäti a ktoré na disku. Teda veľkosť dostupnej fyzickej pamäte neobmedzuje priamo, koľko pamäte môže alokovať jediný proces. Na počítači s malým množstvom fyzickej pamäte jednoducho musí systém virtuálnej pamäte udržovať väčšiu časť naalokovanej pamäte na disku. V dôsledku toho menšieho množstvo fyzickej pamäte znamená väčšie množstvo výpadkov stránok hlavnej pamäte.

preklad adresy a tabuľka stránok: Keď proces adresuje nejaké dáta, musí sa virtuálna adresa preložiť na fyzickú adresu. Keďže hlavná pamäť je zdielaná medzi viacerými programami, spravidla obsahuje iba podmnožinu celého virtuálneho priestoru adresovateľného daným procesom. Takže dáta, na ktoré ukazuje virtuálna adresa, môžu byť v skutočnosti umiestnené na disku. Každý proces preto má tabuľku stránok obsahujúcu mapovanie z virtuálnych adries na fyzické adresy. Tabuľka stránok obsahuje informácie o tom, ktoré virtuálne stránky sa momentálne nachádzajú v hlavnej pamäti a ktoré sa nachádzajú na disku. Samotné tabuľky stránok sú umiestnené v hlavnej pamäti, ale tak ako ktorákoľvek časť hlavnej pamäte, aj tabuľka stránok môže byť dočasne presunutá na disk, ak sa tak rozhodne operačný systém.

Translation Lookaside Buffer (TLB): Tak ako tabuľka stránok, TLB obsahuje mapovanie z virtuálnych na fyzické adresy. Slúži ako cache pre tabuľky stránok tým, že obsahuje najposlednejšie preklady adries. TLB je spravidla implementovaný hardwarovo, aby sa zabezpečilo, že často odkazované virtuálne stránky (kvôli princípu temporal locality) sa preložia čo možno najrýchlejšie.

Keď procesor adresuje nejaké dáta, virtuálna adresa dát sa hľadá zároveň v TLB aj v L1 cache. Level1 cache kontroluje, či obsahuje blok s požadovanou virtuálnou adresou, TLB prekladá virtuálnu adresu na fyzickú adresu, ktorú dodá L1 cache. Ak nastane hit v L1 cache na virtuálnu adresu, potom L1 cache použije fyzickú adresu, ktorú dostalo od TLB, aby skontrolovalo či daný blok je ten správny, alebo či náhodou nepatrí inému programu, s ktorým náhodou zdieľajú rovnakú virtuálnu adresu.

2.2 Latencia pamäte

Pojem latencia pamäte sa používa na popísanie režijných nákladov spojených s pamäťovým systémom. Ale čo je v skutočnosti latencia pamäte, ako ovplyvňuje návrh pamäťových systémov?

Latencia je časový úsek, ktorý strávi jeden komponent čakaním na iný komponent. Inými slovami, premrhaný čas. Teda pamäťová latencia znamená čas procesora premrhaný čakaním na pamäťový systém. Dopad latencie pamäte je často vyjadrovaný v počte cyklov procesora, ktoré procesor stráca čakaním na pamäťový systém. To je praktické z dvoch dôvodov:

1. Taktovacia frekvencia procesora a pamäťového systému sa málokedy zhoduje. Väčšinou je taktovacia frekvencia pamäťového systému menšia ako taktovacia frekvencia procesora. Jeden typ procesora sa často vyrába s rôznymi taktovacími frekvenciami procesora, ale iba s jednou taktovaciou frekvenciou pamäte. Takže aj keď meranie latencie v cykloch frekvencie pamäte je nezávislé od procesora, nie je veľmi prakticky použiteľné. Latencia pamäte je zaujímavá vo vzťahu k výkonu procesora.

2. V závislosti na stupni paralelizmu na úrovni inštrukcií sú moderné procesory schopné vykonať niekoľko inštrukcií behom jedného cyklu. Ak poznáme tento stupeň paralelizmu a latenciu pamäte v cykloch procesora, môžeme bez problémov interpretovať dopad latencie jednoducho spočítaním počtu inštrukcií, ktoré sa mohli vykonať, kým procesor čakal na pamäťový systém.

Latencia pamäte však nie je nemenná metrika. Je špecifická pre danú úroveň pamäťovej hierarchie a zvyšuje sa čím sme ďalej od procesora. Preto každá vrstva pamäťovej hierarchie pridáva k latencii celého pamäťového systému. Za predpokladu, že dáta sa nenačítavajú paralelne, pre každú vrstvu je latencia suma času stráveného hľadaním požadovaného bloku a času stráveného presúvaním tohto bloku do rýchlejšej vrstvy. Preto požiadavka na dáta, ktorá spôsobí výpadky cache naprieč celou pamäťovou hierarchiou, spôsobí latenciu odpovedajúcu sume latencií všetkých pamäťových vrstiev.

Čas, ktorý procesor strávi čakaním na pamäťový systém, nie je niekedy celkom stratený. Ak inštrukcia i spôsobí výpadok cache a vykonávanie nasledujúcej inštrukcie j nezávisí od výsledku i, procesor môže vykonať inštrukciu j, kým bude čakať na pamäťový systém.

Latencia spôsobená konkrétnou vrstvou pamäťovej hierarchie je ovplyvnená asociativitou danej vrstvy pamäťovej hierarchie. Čo do vrstiev pamäťovej hierarchie najbližšie k procesoru, asociativita týchto vrstiev je implementovaná hardwarovo a vyšší stupeň asociativity znamená zložitejšie obvody a preto vyššiu latenciu. Úbytok výpadkov cache spôsobený vyššou asociativitou nevyrovná nárast latencie pamäťového systému. Na základe výsledkov z [6, s. 11], ktoré hovoria, že od určitej veľkosti cache pamäte má stupeň asociativity minimálny dopad na počet cache výpadkov, dochádzame k záveru, že nárast latencie pamäte znemožňuje použitie plnej asociativity vo vrstvách najbližšie k procesoru.

Latenciu môžu samozrejme spôsobovať aj iné časti počítaču ako pamäťový systém. Napr. riziko zlej predikcie skoku alebo iné závislosti medzi inštrukciami môžu pozdržať procesor na nejaký počet cyklov alebo spôsobiť úplné vyprázdnenie pipeliny procesora. Latenciu tiež spôsobuje komunikácia s externými zariadeniami (grafickými, sieťovými).

2.3 Praktická ukážka pamäťovej latencie a výpadkov cache pamäte

Z predchádzajúcich sekcií by už malo byť jasné, že zohľadnenie latencie pamäte a prítomnosti cache pamäte pri navrhovaní algoritmu môže pozitívne ovplyvniť čas jeho behu v praxi. Ale ako malé či veľké sú v skutočnosti následky neopatrného návrhu algoritmu? Ako v skutočnosti nastávajú rôzne druhy cache výpadkov? Na tieto otázky odpovieme použitím jednoduchého príkladu.

V [26] Bojesen a spol. implementovali jednoduchý program za účelom sledovania L1 cache pamäte na rôznych počítačoch. Avšak my môžeme tento program použiť inak ako bolo pôvodne zamýšľané. Pomocou tohto programu budeme zisťovať, ako beží program navrhnutý v súlade s princípmi pamäťovej hierarchie oproti programu, ktorý využíva pamäťovú hierarchiu neefektívne.

Program počíta sumu N integerov (celých čísel) uložených súvisle v poli. V závislosti na hodnote premennej *krok* sa na prvky v poli pristupuje podľa rôznych vzorov:

```
unsigned int sum(unsigned int* pole,
                 unsigned int krok,
                 unsigned int N) {
      unsigned int i;
      unsigned int maska = N - 1;
      unsigned int result = pole[0];

      for( i = krok; i != 0; i = (i + krok) & maska)
            result += pole[i];

      return result;
}
```

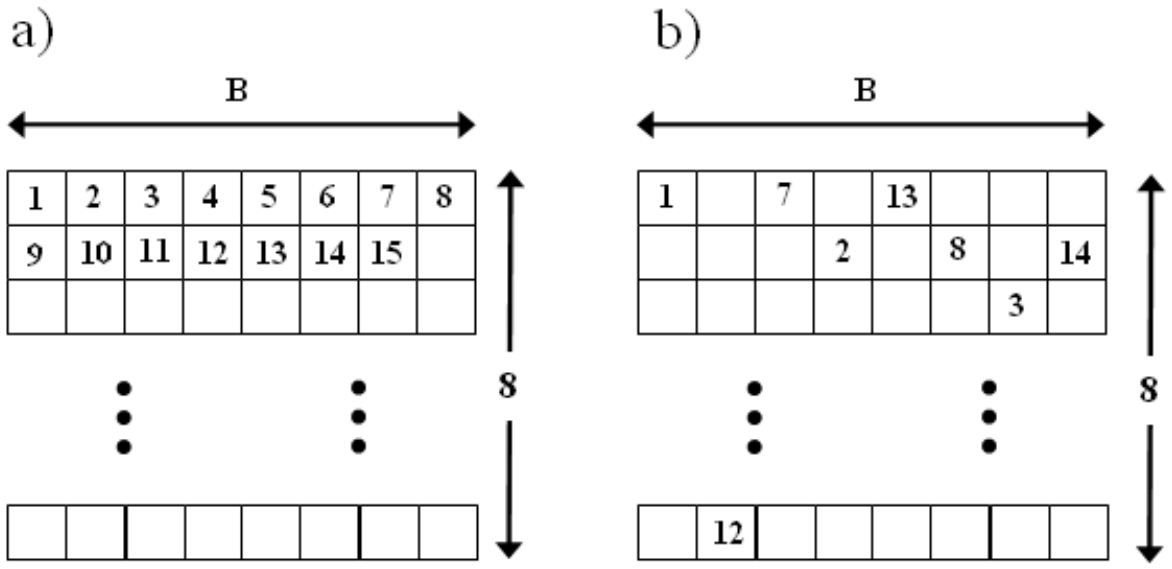
Keď je hodnota premennej $krok = 1$, na prvky sa pristupuje sekvenčne. Keď je hodnota premennej $krok = p$, kde p je najmenšie prvočíslo väčšie ako veľkosť bloku, na prvky v poli sa pristupuje tak, že žiadne dva po sebe nasledujúce prístupy nebudú vykonané do jedného cache riadka. Tým, že index *i*-teho navštíveného prvku počítame ako $i * krok \bmod N$ a ďalej tým, že za hodnotu N zvolíme mocniny dvojky, hodnoty premennej *krok* 1 a p zaručujú, že všetky prvky v poli budú navštívené práve jeden krát. Poznamenajme, že rozklad každého celého čísla na prvočísla existuje iba jeden a pre mocniny dvojky je jediným prvočíslom v rozklade číslo dva.

Pre obe hodnoty premennej *krok* vykoná program rovnaký počet inštrukcií, takže rozdiel v čase behu oboch programov je spôsobený výlučne rozdielom v spôsobe prechádzania poľa.

Nech B označuje počet integerov, ktoré sa zmestia do bloku, potom program spôsobí N/B cache výpadkov L1 cache pamäte pre hodnotu premennej $krok = 1$ a N cache výpadkov L1 cache pamäte pre hodnotu premennej $krok = p$. Výsledky Bojesena a spol. [26] ukazujú, že program je tri až desať krát rýchlejší keď spôsobuje iba N/B cache výpadkov ako keď spôsobuje N cache výpadkov L1 cache pamäte. Variácia v relatívnej efektívnosti výsledkov je spôsobená rôznymi veľkosťami bloku a rozdielmi vo veľkosti a asociativite cache pamätí na testovaných počítačoch.

Popísaný príklad sme testovali aj my na počítači s procesorom Intel Celeron D336 (2,8 GHz). Špecifikáciu jeho cache pamätí nájdeme v tabuľke Tab. 2.2, ale pre istotu ich uvedieme aj na tomto mieste – L1 cache: 16+16 kB, 4-cestná množinovo asociatívna, 64-bytové cache riadky; L2 cache: 256 kB, 8-cestná množinovo asociatívna, 128-bytové cache riadky implementované ako dva 64-bytové sektory/bloky. Uskutočnili sme niekoľko meraní, každé meranie malo 30 opakovaní. Výsledný čas sme počítali ako aritmetický priemer 30 opakovaní daného merania. Sledovanou

veličinou bol čas behu programu, konkrétnejšie čas behu funkcie sum(). Meranie bolo uskutočnené pod operačným systémom Gentoo Linux a počas merania nebežal okrem jadra a meraného programu žiadny iný proces (t.j. systém nebol vôbec zaťažený). Viac detailov je možné nájsť v komentároch zdrojových súborov programu, ktoré sa nachádzajú na priloženom cdčku v adresári 'src/ukazka'. Pre veľkosti poľa presahujúce veľkosť L2 cache pamäte, program bežal 18,5 až 20 krát rýchlejšie keď spôsoboval N/B cache výpadkov ako keď spôsoboval N cache výpadkov.



Obr. 2.4: Číslovanie označuje poradie v akom sa pristupuje do cache pamätí, keď je počet prvkov poľa väčši alebo rovný kapacite cache pamäte. **a)** Efektívny prístupový vzor pristupuje do cache pamäte plynule, t.j. s krokom 1. **b)** Neefektívny prístupový vzor pristupuje do cache pamäte s krokom p, kde p je najmenšie prvočíslo väčšie ako veľkosť bloku B.

Na obrázku Obr. 2.4 sú názorne zobrazené prístupové vzory pre dve hodnoty premennej *krok* v zjednodušenej cache pamäti, o ktorej predpokladáme, že je plne asociatívna a každý z jej ôsmich cache riadkov obsahuje jeden blok ôsmich integerov.

Na prvý pohľad nemusí byť zrejmé, prečo je počet cache výpadkov programu N a N/B. Preto v nasledujúcom odstavci dokážeme, že to platí. Tak ako na obrázku Obr. 2.4, predpokladáme plne asociatívnu cache pamäť, stratégiu výmeny LRU a kapacitu cache pamäte označíme ako M:

krok = 1: Na začiatku je cache pamäť prázdna. Takže kým sa dáta zmestia do cache pamäte, t.j. $N \leq M$, sekvenčné pristupovanie do poľa spôsobí compulsory výpadok pre každých B prístupov. Takže celý program spôsobí N/B cache výpadkov keď je $N \leq M$.

Ak $N > M$, vykoná sa ďalších $N - M$ prístupov. Pretože cache pamäť neobsahuje už žiadne prázdne riadky, tieto dodatočné prístupy spôsobia jeden capacity cache výpadok pre každých B prístupov, čo je vo výsledku $(N - M)/B$ capacity cache výpadkov.

Ak nebudeme rozlišovať medzi typmi cache výpadkov, t.j. nebudeme rozlišovať či nastal compulsory alebo capacity cache výpadok, potom počet cache výpadkov programu je N/B bez ohľadu na veľkosť vstupného poľa.

krok = p: Hodnotu čísla p volíme ako najmenšie prvočíslo väčšie ako veľkosť bloku, aby sme zaručili, že žiadne dva po sebe nasledujúce prístupy nepristúpia do toho istého cache riadka. Pokiaľ sa veľkosť vstupného poľa zmestí do cache pamäte, program spôsobí iba N/B cache výpadkov. Oproti prvému prípadu, kedy program spôsobil cache výpadok pre každých B prístupov, teraz program spôsobuje compulsory cache výpadok pre prvých N/B prvkov. Prístupy na zvyšných $N - N/B$ prvkov nespôsobia žiadne ďalšie cache výpadky.

Ak veľkosť vstupných dát presiahne kapacitu cache pamäte, situácia sa trochu skomplikuje. Najskôr si všimneme, že po prvých M/p prístupoch je cache pamäť plná a nasledujúci prístup bude vykonaný do bloku, ktorý predtým ešte nebol v cache pamäti (pozn.: teraz je $N > M$). Teda nastane capacity cache výpadok a v súlade so stratégiou výmeny LRU spôsobí vyhodenie bloku, ktorý bol referencovaný ako prvý.

V závislosti na veľkosti N môže pre ďalší prístup nastať jedna z dvoch situácií. Buď sa ďalší prvok nachádza v bloku, ktorý ešte nebol predtým v cache pamäti (tzn. ešte sme nedosiahli koniec poľa), alebo sme dosiahli index presahujúci veľkosť N (tzn. sme znova na začiatku poľa). Obe z týchto dvoch situácii spôsobia výpadok cache pamäte. V prvej z možných situácií nastane capacity cache výpadok tak ako pri predchádzajúcom prvku (vyhodí sa blok v súlade so stratégiou výmeny LRU). V druhej z možných situácií tiež nastane capacity cache výpadok, pretože blok na ktorý chceme pristúpiť (t.j. blok obsahujúci začiatok poľa) bol práve z cache pamäte vyhodený. To znamená, že pre $N > M$, každý prístup spôsobí cache výpadok čím dostávame počet cache výpadkov programu rovný N.

V závislosti na hodnote premennej *krok* bol program spustený s cieľom spôsobiť čo najviac alebo čo najmenej cache výpadkov, takže časy behu programu predstavujú extrémne prípady práce s cache pamäťou. Ale aj programátor, ktorý si nie je vedomý dopadu pamäťovej latencie, málokedy vytvorí program, ktorý sa chová až tak zle ako príklad našeho programu pre *krok* = p. Aj napriek tomu, tento príklad jasne ukázal, že pri návrhu programu je treba dbať pamäťovej latencie, nakoľko môže urýchliť beh programu o konštantný faktor, ktorý nie je zanedbateľný. Analýza programu v modeli RAM by ilustrované správanie programu neodhalila.

2.4 Moderné pamäťové systémy (SMP, NUMA)

Procesory sú dnes oveľa sofistikovanejšie ako boli pred tridsiatimi rokmi. Vtedy bola frekvencia jadra procesora na ekvivalentnej úrovni s frekvenciou pamäťovej zbernice. Prístup do pamäte bol iba o málo pomalší ako prístup do registra. Dramatická zmena prišla na začiatku deväťdesiatych rokov, kedy vývojári hardwaru zvýšili frekvenciu jadra procesora, ale frekvencia pamäťovej zbernice a výkon pamäťových čipov sa primerane nezvýšili. Nestalo sa tak preto, že by sa nedali vyrobiť rýchlejšie čipy RAM, ale z dôvodu, že by to bolo neekonomické.

Ak by sme si mohli vybrať medzi počítačom s malou, ale veľmi rýchlou pamäťou RAM, alebo počítačom s veľkým množstvom relatívne rýchlej pamäte RAM, druhá možnosť je vždy lepšia, keď uvážime veľkosť spracovávaných dát, ktorá presahuje veľkosť malej pamäte a cenu za prístup do zálohovacích médií ako napr. disky. Problémom je rýchlosť zálohovacieho média, ktoré sa musí použiť pre uloženie časti spracovávaných dát, ktoré sa do pamäte nezmestili. Prístup na

zálohovacie médium je vždy rádovo pomalší ako prístup do pamäte.

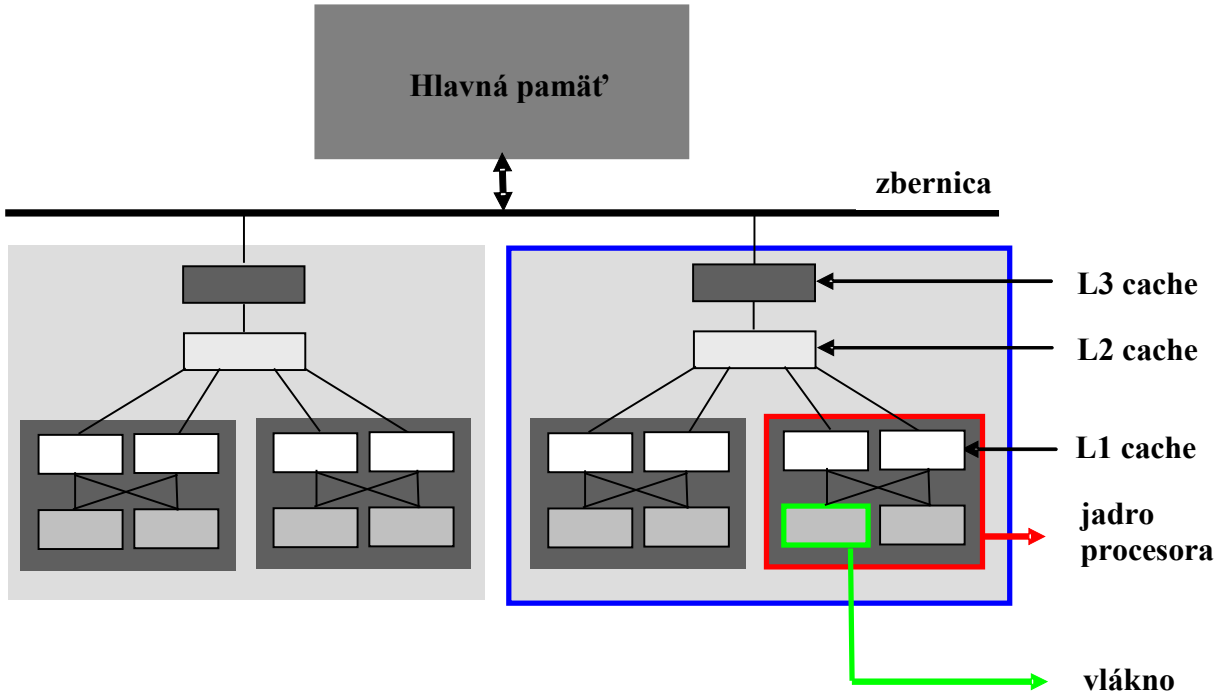
Našťastie sa nemusíme rozhodovať len medzi týmito dvomi možnosťami. Počítač môže mať malé množstvo veľmi rýchlej pamäte SRAM doplnenej veľkým množstvom pamäte DRAM. Jedna z možných implementácií by bola vyhradiť určitú časť adresového priestoru procesora pre SRAM a zvyšok pre DRAM. Úlohou operačného systému by potom bolo optimálne rozdeľovať dáta tak, aby sa využila pamäť SRAM, ktorá by slúžila ako rozšírenie sady registrov.

Toto riešenie je síce možné, ale neschodné. Aj keby sme ignorovali problém mapovania fyzických prostriedkov takejto pamäte na virtuálny adresový priestor procesov, tento prístup by vyžadoval, aby si každý proces softwarovo spravoval alokáciu danej pamäťovej oblasti. Veľkosť pamäťovej oblasti je u rôznych procesorov rôzna. Všetky moduly programu si zaberú svoju časť rýchlej pamäte, čím nám ale vzniká potreba synchronizácie a s ňou súvisiace náklady. Zisk plynúci z rýchlej pamäte by bol úplne spotrebovaný režijnými nákladmi na administráciu zdrojov. Teda namiesto toho, aby SRAM bola pod kontrolou operačného systému alebo užívateľa, spraví sa z nej zdroj, ktorý je transparentne používaný a spravovaný procesorom. Týmto spôsobom je SRAM využívaná na vytváranie dočasných kópií (do cache) dát v hlavnej pamäti, o ktorých sa predpokladá, že sa v blízkej dobe použijú.

Veľkosť SRAM pamäte použitej pre cache pamäť je v drvivej väčšine prípadov menšia ako veľkosť hlavnej pamäte. Pomer veľkosti cache pamäte k hlavnej pamäti býva aj okolo jedna ku tisíc (dnes sa vyrábajú počítače s 4 až 6MB cache pamäťou a 4GB hlavnou pamäťou). Počítače ale nemajú veľké množstvá hlavnej pamäte bez dôvodu. Veľkosť spracovávaných dát je spravidla vačšia ako cache. Zvlášť to platí pre systémy, kde beží naraz viac procesov a veľkosť spracovávaných dát je súhrn veľkostí všetkých procesov plus jadra systému.

Onedlho po zavedení cache pamätí sa počítačové systémy ešte viac skomplikovali. Rozdiel v rýchlosti hlavnej pamäte a cache sa zväčšil až natoľko, že bola pridaná ďalšia úroveň cache pamäte (väčšia a pomalšia ako cache prvej úrovne). Zväčšenie cache pamäte prvej úrovne nepripadalo do úvahy z ekonomických dôvodov. Dnes existujú dokonca počítače s tromi úrovňami cache pamätí.

Naviac existujú procesory, ktoré majú viacero jadier a každé jadro môže mať viacero *threadov* (*vlákien*). Rozdiel medzi jadrom a vláknom je, že jadrá majú oddelené kópie skoro všetkých hardwarových zdrojov. Jadrá môžu pracovať nezávisle, pokiaľ nepoužívajú tie isté zdroje v tom istom čase. Na druhej strane vlákna zdieľajú skoro všetky zdroje procesora. Schéma moderného procesora je zobrazená na obrázku Obr. 2.5 a na obrázku Obr. 2.6.

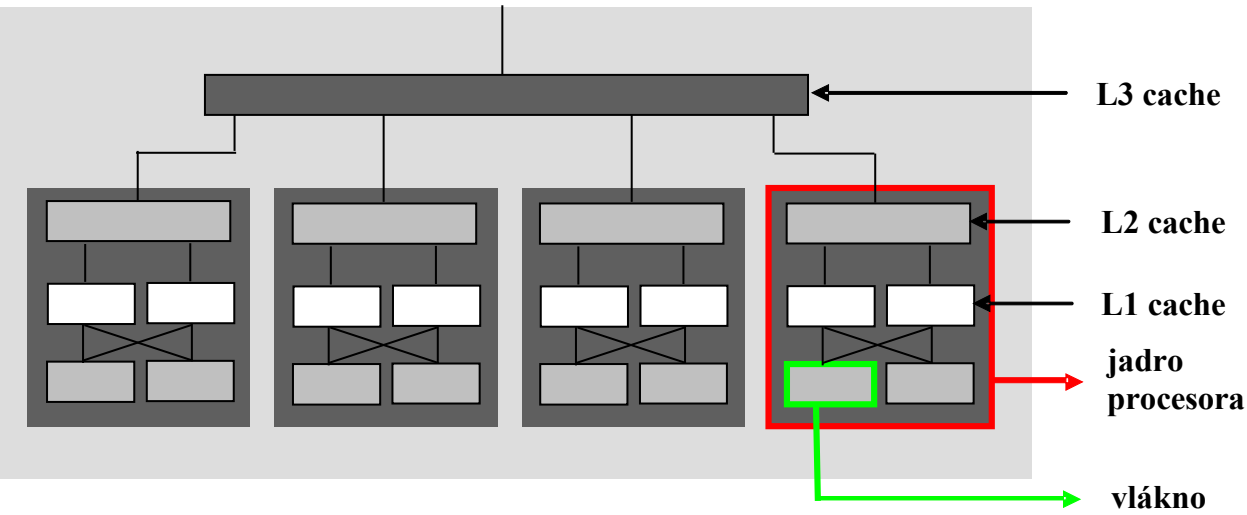


Obr. 2.5: Multi procesor, multijadrový, multivláknový.

Na obrázku Obr. 2.5 sú zobrazené dva procesory, každý s dvomi jadrami a každé jadro má dve vlákna. Vlákna zdieľajú L1 cache pamäte. Jadrá majú vlastné L1 cache pamäte, ale všetky cache pamäte vyššej úrovne zdieľajú. Procesory navzájom samozrejme nezdieľajú žiadne cache pamäte. Druhý príklad moderného procesora je zobrazený na obrázku Obr. 2.6. Obsahuje štyri jadrá a každé jadro má dve vlákna. Vlákna zdieľajú L1 cache pamäte. Jadrá majú vlastné L1 a L2 cache pamäte a zdieľajú až L3 cache pamäť.

V SMP (symmetric multi processor) systémoch nemôžu cache pamäte jednotlivých procesorov pracovať nezávisle od seba. Všetky procesory by mali vždy vidieť rovnaký obsah pamäte v rovnakom stave. Vlastnosť jednotného vnímania pamäte z pohľadu procesorov sa nazýva koherentnosť cache pamätí. Ak by sa procesor vždy díval iba na obsah svojich cache pamätí a obsah hlavnej pamäte, nevidel by obsah cache riadkov ostatných procesorov, označených príznakom „dirty“. Priamy prístup procesora ku cache pamätiam iného procesora by bol nákladný. Namiesto toho sa to robí tak, že procesory zisťujú prípad, keď iný procesor chce čítať alebo zapisovať na nejaký cache riadok (zjednodušene povedané).

Ak sa detekuje zápis a procesor má validnú kópiu cache riadka vo svojej cache pamäti, tento riadok sa označí ako invalidný. Budúce adresovanie tohto riadka bude vyžadovať opätovné načítanie. Čítanie na inom procesore neznamená nutnosť invalidácie daného cache riadka. Viacero validných kópií cache riadka môže bez problémov koexistovať.



Obr. 2.6: Procesor, multijadrový, multivláknový.

Sofistikovanejšie implementácie cache pamätí vnášajú do problému ďalšiu možnosť. Ak je cache riadok, ktorý chce iný procesor čítať alebo do ktorého chce zapisovať, momentálne označený v cache pamäti prvého prosesora ako dirty, je potrebný iný postup. V tomto prípade neobsahuje hlavná pamäť aktuálne dáta a procesor, ktorý inicioval čítanie alebo zápis, musí získať obsah požadovaného cache riadka od prvého procesora. Pomocou techniky snoopingu (technika, kedy každý radič cache pamäte monitoruje na zbernici prístupy na pamäťové miesta, ktoré obsahuje daná cache pamäť) vie prvý procesor zistiť, že táto situácia nastala a automaticky pošle druhému procesoru požadované dáta. Tento prenos obchádza hlavnú pamäť, ale v niektorých implementáciách ho radič pamäte vie detekovať a uloží aktualizovaný cache riadok do hlavnej pamäte. Ak táto situácia nastala pre zápis, prvý procesor následne zneplatní svoju kópiu cache riadka.

V minulosti bolo vyvinutých veľa protokolov na udržovanie koherentnosti cache pamäte. Najdôležitejší z nich je MESI protokol, ktorý je popísaný v Sekcii 2.4.2. Ponaučenie z predchádzajúceho príkladu sa dá zhrnúť do dvoch pravidiel:

1. Cache riadok označený ako dirty sa nenachádza v cache pamäti žiadneho iného procesora.
2. Validné kópie cache riadka sa môžu nachádzať v cache pamätiach ľubovoľne veľa procesorov.

Keď sa dodržia tieto pravidlá, procesory môžu používať svoje cache pamäte efektívne aj v multi procesorových systémoch. Jediné, čo procesory musia robiť, je monitorovať vzájomné prístupy na zápis a porovnávať adresy so svojimi lokálnymi cache pamäťami.

2.4.1 Write policy

Cache pamäte majú byť koherentné a táto koherentnosť má byť programátorovi úplne transparentná. To znamená, že ak nastane zmena obsahu cache riadka, výsledok je pre systém

rovnaký ako keby neexistovala žiadna cache pamäť a zmena bola vykonaná priamo v hlavnej pamäti. Toto je možné dosiahnuť dvomi stratégiami (neskôr sú uvedené ešte ďalšie dve):

- write-through cache implementácia
- write-back cache implementácia

Najjednoduchšie implementovateľná je write-through stratégia. Ak sa zapíše do cache riadka, procesor ihneď zapíše cache riadok aj do hlavnej pamäte. To zaručí, že cache pamäť a hlavná pamäť sú vždy synchronizované. Obsah cache riadka sa pri nahradení proste zahodí. Táto stratégia je jednoduchá ale pomalá.

Stratégia write-back je sofistikovanejšia. Na rozdiel od write-through implenetácie, keď sa zapíše do cache riadka, procesor nezapisuje cache riadok do hlavnej pamäte. Namiesto toho označí riadok ako dirty. Keď sa potom cache riadok bude vyhadzovať z cache pamäte, príznak dirty povie procesoru, že má obsah zapísať do hlavnej pamäte.

Táto stratégia má ale jeden vážny problém. Keď je k dispozícii viac ako jeden procesor (alebo jadro, hyperthread) a pristupuje sa na tú istú pamäť, musí byť zaručené, že všetky procesory vidia ten istý obsah pamäte. Ak je cache riadok označený príznakom dirty na jednom procesore (t.j. ešte nebol zapísaný do hlavnej pamäte) a druhý procesor sa snaží prečítať to isté pamäťové miesto, operácia čítania nemôže proste siahnuť do hlavnej pamäte. Požadované dáta sa nachádzajú v cache pamäti prvého procesoru. V ďalšej sekcii si ukážeme ako je to v implementované v súčasnosti.

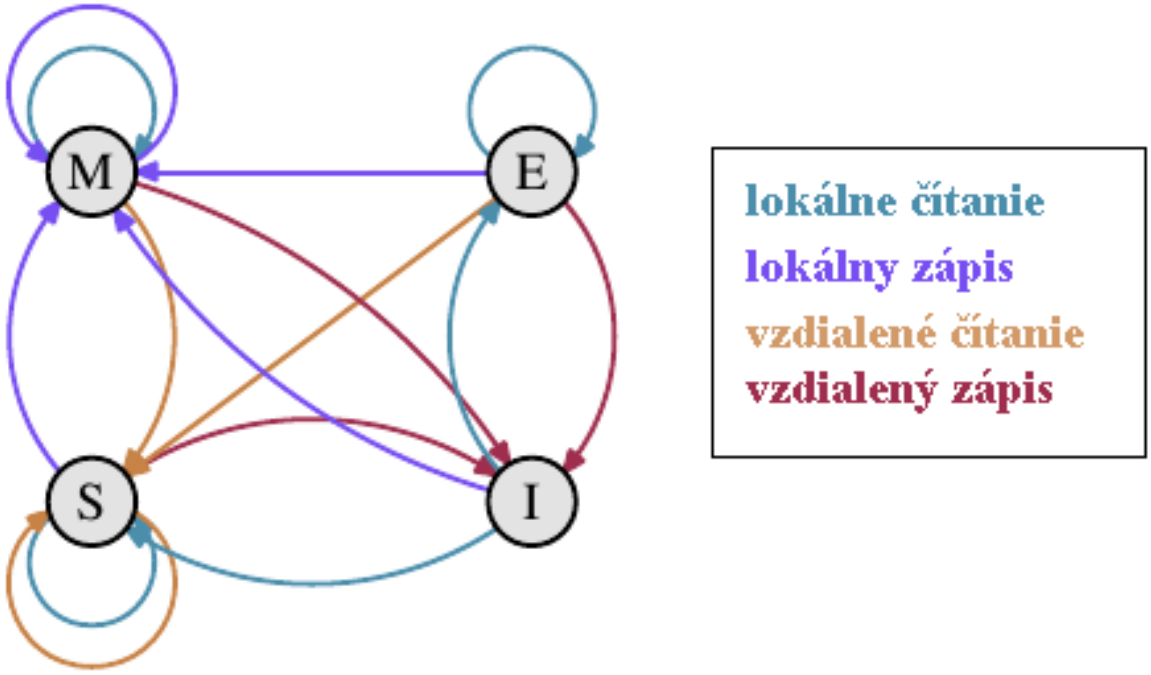
Ako sme uviedli na začiatku sekcie, existujú ešte ďalšie dve stratégie, konkrétne write-combining a uncacheable. Tieto dve stratégie sa ale používajú pre špeciálne oblasti adresového priestoru, ktorý nemá za sebou skutočnú RAM pamäť. Z tohto dôvodu sa im nebudeme venovať.

2.4.2 Multiprocesorová podpora

V predchádzajúcej sekcií sme načrtli problém, ktorý vzniká keď uvažujeme viacero procesorov. Tento problém majú aj viacjadrové procesory pre tie úrovne cache pamätí, ktoré nie sú zdieľané (čo je vždy aspoň L1 data cache). Bolo by prinajmenšom nepraktické poskytovať priamy prístup z jedného procesora do cache pamäte druhého procesora, to spojenie jednoducho nie je dostatočne rýchle. Praktickejšou alternatívou je prenášať obsah cache pamäte do druhého procesora v prípade, keď je to potrebné. Na otázku, kedy by mal prenos cache riadkov nastávať, existuje jednoduchá odpoveď. Vtedy, keď procesor potrebuje čítať alebo zapisovať na cache riadok, ktorý je označený príznakom dirty v cache pamäti iného procesora. Ako ale môže procesor zistiť, či je nejaký cache riadok dirty v cache pamäti iného procesora? Predpokladať, že cache riadok je dirty v cache pamäti iného procesora preto, že iný procesor daný cache riadok načítal, je nesprávne. Väčšina pamäťových prístupov sú prístupy pre čítanie a tie nespôsobujú nastavenie príznaku dirty. A posielať informácie o zmenených cache riadkoch po každom zápise je minimálne nepraktické, nakoľko operácie procesora nad cache riadkami sú časté. Na otázku: „Tak ako na to?“, odpovie nasledujúci odstavec.

Ako sme už hovorili, najdôležitejším protokolom na udržovanie koherentnosti cache pamäte je MESI (Modified, Exclusive, Shared, Invalid) protokol. Je pomenovaný po štyroch stavoch, v ktorých sa môže nachádzať cache riadok:

●*modified*: Lokálny procesor pozmenil cache riadok. To implikuje, že tento riadok je jedinou kópiou vo všetkých cache pamätiach.
●*exclusive*: Cache riadok nie je pozmenený (teda odpovedá hlavnej pamäti) a nenachádza sa v žiadnej cache pamäti iného procesora.
●*shared*: Cache riadok nie je pozmenený a môže sa nachádzať v cache pamäti iného procesora.
●*invalid*: Cache riadok je neplatný, t.j. nepoužívaný.



Obr. 2.7: Prechody medzi stavmi protokolu MESI (**Modified**, **Exclusive**, **Shared**, **Invalid**).

Tento protokol sa postupom času vyvinul z jednoduchších verzií. S popísanými štyrmi stavmi je možné efektívne implementovať write-back cache pamäte a zároveň umožniť súbežné čítanie dát na rôznych procesoroch. Zmeny stavov protokolu MESI sú zobrazené na Obr. 2.7:
●lokálne čítanie: E → E, I → E, I → S, S → S, M →M
●lokálny zápis: I → M, E → M, M → M, S →M
●vzdialené čítanie: S → S, E → S, M → S
●lokálny zápis: S → I, M → I, E → I

Zmeny stavov sa dosiahnu tak, že procesor monitoruje (technika bus-snooping) prácu iných procesorov. Niektoré operácie procesora sa ohlásia na externých pinoch a tým zviditeľňujú prácu tohto procesora iným procesorom. Adresy cache riadkov sú viditeľné na adresovej zbernici.

Na začiatku sú všetky cache riadky prázdne a teda v stave *invalid*. Ak sa do cache načítajú dáta pre operáciu zápisu, zmení sa stav cache riadka na *modified*. Ak sa dáta do cache načítajú pre operáciu čítania, stav riadka závisí od toho, či je tento riadok načítaný aj v cache pamäti iného procesora. Ak áno, je nový stav cache riadka *shared*, v opačnom prípade je stav cache riadka *exclusive*.

Ak je cache riadok, ktorý je označený ako *modified*, načítavaný z lokálneho procesora alebo je do tohto riadka zapisované, môže sa použiť súčasný obsah cache riadka bez zmeny stavu. Ak chce iný procesor čítať daný cache riadok, lokálny procesor musí poslať obsah cache riadka druhému procesoru a následne môže lokálny procesor zmeniť stav cache riadka na *shared*. Dáta poslané druhému procesoru sú spracované aj radičom pamäte, ktorý uloží dáta do hlavnej pamäte. Ak chce iný procesor zapisovať do cache riadka lokálneho procesora, lokálny procesor pošle obsah cache riadka druhému procesoru a označí svoj cache riadok ako *invalid*. Deje sa tak prostredníctvom RFO (Request For Ownership) operácie, ktorá je relatívne časovo náročná.

Ak sa cache riadok nachádza v stave *shared* a lokálny procesor z neho číta, nenastáva zmena stavu a požiadavok sa môže uspokojiť z lokálnej cache pamäte. Ak sa lokálne zapisuje do cache riadka označeného ako *shared*, môže sa daný cache riadok použiť, ale jeho stav sa zmení na *modified* a všetky potenciálne kópie cache riadka v cache pamätiach iných procesorov zmenia svoj stav na *invalid*. Preto sa musia operácie zápisu oznamovať procesorom cez RFO správy. Ak iný procesor požaduje cache riadok pre operáciu čítania, nie je potrebné spraviť nič, hlavná pamäť obsahuje súčasné dáta a stav cache riadka je už označený ako *shared*. V prípade, keď chce iný procesor zapisovať do cache riadka, pošle lokálnemu procesoru RFO správu a stav cache riadka sa zmení na *invalid*.

Stav *exclusive* sa najviac podobá stavu *shared* s jedným podstatným rozdielom. Lokálny zápis do cache riadka sa nemusí ohlasovať na zbernici – lokálna kópia cache riadka je jediná existujúca kópia cache riadka. Z časového hľadiska je to veľká výhoda a každý procesor sa snaží udržovať čo najviac cache riadkov v stave *exclusive* namiesto stavu *shared*. Stav *exclusive* by sa v podstate mohol úplne vynechať bez zmeny funkčnosti alebo správnosti. V takom prípade by ale utrpela výkonnosť, nakoľko prechod E → M je oveľa rýchlejší ako prechod S → M.

Z popisu protokolu sa ukazuje, kde sú najväčšie náklady špecifické pre SMP. Sú to naplnenie cache pamätí a RFO správy. RFO správy sú nevyhnutné v dvoch situáciách. Prvá situácia nastáva, ak sa presúva vlákno z jedného procesora na druhý a musia sa naraz presunúť všetky cache riadky z cache pamäte jedného procesora do cache pamäte druhého procesora. Druhá situácia nastáva, ak sa cache riadok naozaj musí nachádzať v cache pamätiach dvoch rôznych procesorov. V menšom merítku to platí aj pre viac jadier v jednom procesore.

V multi-vláknových alebo multi-procesových programoch je vždy nutná nejaká synchronizácia. Táto synchronizácia je implementovaná použitím pamäte. Čiže niektoré RFO správy sú nevyhnutné, ale aj tak je treba udržovať ich počet čo najmenší. Správy RFO posielané akýmkoľvek protokolom spravujúcim koherenciu cache pamäte musia byť rozdelené všetkým procesorom systému. Zmena stavu v MESI protokole nemôže nastať skôr, ako je zaručené, že všetky procesory systému mali možnosť na správu reagovať. To znamená, že rýchlosť protokolu pre správu koherentnosti cache pamäte je určená tým, ako najdlhšie môže trvať odpoveď. Kolízie na zbernici nie sú vylúčené, latencia v NUMA systémoch môže byť veľmi veľká a samozrejme samotný objem toku dát môže spomaľovať celý systém – samé podstatné dôvody, prečo sa vyvarovať zbytočnému toku dát/informácií.

Ďalším problémom viacprocesorových systémov je, že zbernica front-side bus je zdieľaná. Na veľkej väčšine počítačov sú všetky procesory pripojené k radiču pamäte prostredníctvom jedinej zbernice. Ak jediný procesor v jednoprocesorovom systéme dokáže túto zbernicu zahltiť, potom dva a viacej procesorov, zdieľajúcich túto zbernicu, obmedzia priepustnosť zbernice pre každý procesor ešte výrazne viac. Aj v prípade, ak má každý procesor vlastnú zbernicu k radiču pamäte,

limitujúcim prvkom je zbernica vedúca k pamäťovým modulom. Väčšinou je to jedna zbernica, ale aj keby nebola, súčasné prístupy do rovnakého pamäťového modulu obmedzujú priepustnosť.

Záver je, že programy, ktoré majú efektívne bežať na SMP, musia byť vhodne navrhnuté, aby minimalizovali prístupy na rovnaké pamäťové miesta z rozdielnych procesorov a jadier procesorov.

2.4.3 NUMA

Ako sme videli v predchádzajúcej sekcií, v typickej SMP architektúre všetky prístupy do pamäte využívajú zdieľanú pamäťovú zbernicu. To funguje dobre pre malý a obmedzený počet procesorov. Problém so zdieľanou pamäťovou zbernicou ale nastane, keď na ňu zapojíme veľký počet procesorov, ktoré sa bijú o prístup. Architektúra NUMA bola navrhuntá s cieľom:

- Adresovať tento problém poskytnutím samostatnej pamäte pre každý procesor.
- Prekonať obmedzenie rozšíriteľnosti SMP. Numa architektúra poskytuje rozšíriteľnosť rozsahu MPP(Massively Parallel Processing) v tom zmysle, že procesory môžeme pridávať a odoberať bez straty efektivity.

NUMA je skratka pre non-uniform memory access. To znamená, že prístup na niektoré pamäťové oblasti trvá dlhšie. Je to spôsobené tým, že niektoré pamäťové oblasti sa nachádzajú na fyzicky iných zberniciach. Následkom toho môžu programy, ktoré nie sú pre túto architektúru navrhnuté, bežať drasticky pomalšie.

Hlavným rozdielom medzi architektúrou NUMA a SMP je, že NUMA bola navrhnutá s cieľom prekonať obmedzenie rozšíriteľnosti SMP. Architektúra NUMA sa s týmto problémom vysporiadava tak, že obmedzuje počet procesorov, ktoré môžu byť pripojené na jednu pamäťovú zbernicu a jednotlivé *uzly* spája rýchlym prepojením.

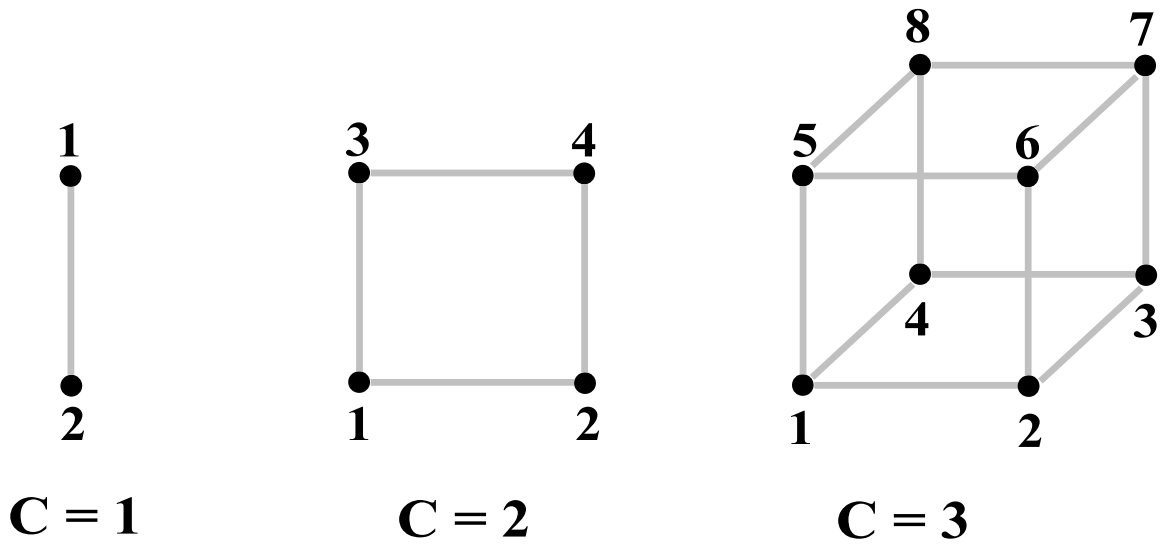
Jedným z problémov pri popisovaní architektúry NUMA je, že existuje mnoho spôsobov ako túto technológiu implementovať. To viedlo k vzniku množstva definícií pre *uzol*. Po technickej stránke správna definícia uzla je: región pamäte, v ktorom má každý byte rovnakú vzdialenosť od každého procesora. Rozšírenejšia je ale menej formálna definícia, ktorá definuje uzol ako región pamäte, procesor/y, vstupno/výstupné zariadenia atď., ktoré sa fyzicky nachádzajú na jednej zbernici. Niektoré architektúry ale nemajú pamäť, všetky procesory, a vstupno/výstupné zariadenia fyzicky na jednej jedinej zbernici a v takom prípade menej formálna definícia neplatí.

S architektúrou NUMA sa nevyhnutne spája pojem lokálnej a vzdialenej pamäte. Pojmy lokálna a vzdialená pamäť sa väčšinou používajú v súvislosti s práve bežiacim procesom. Lokálna pamäť sa väčšinou definuje ako pamäť nachádzajúca sa na rovnakom uzle ako procesor práve vykonávajúci daný proces. Každá pamäť, ktorá nepatrí uzlu, na ktorom sa práve vykonáva daný proces, je vzdialená. Pojem lokálna a vzdialená pamäť sa tiež môže používať s vecami nesúvisiacimi s práve vykonávaným procesom. Keď sa nachádzame v kontexte prerušenia, technicky neexistuje žiadny práve vykonávaný proces, ale pamäť na uzle obsahujúcom procesor obsluhujúci prerušenie sa aj tak nazýva lokálna. Pojmy lokálna a vzdialená pamäť sa tiež používajú v súvislosti s diskom – napr. keď budeme mať disk (pripojený k uzlu 1) vykonávajúci priamy prístup do pamäte (DMA), pamäť ktorú číta alebo do ktorej zapisuje sa nazýva vzdialená, ak sa nachádza na inom uzle (pr. uzol 0). V architektúrach NUMA ďalej potrebujeme nejakým spôsobom popísať vzdialenosti medzi komponentami systému. Používajú sa rôzne metriky ale najznámejšou

sú hopy (angl. hops).

Architektúra NUMA má však aj svoje úskalia. Napríklad procesory na nejakom uzle majú väčší prietok dát a/alebo nižšiu latenciu pre prístup k pamäti a ostatným procesorom na tom istom uzle. Kvôli tomu môžu nastávať situácie ako starvation lock – ak procesor x na danom uzle požaduje zámok, ktorý už drží iný procesor y na tom istom uzle, požiadavky procesora x majú väčšinou tendenciu prebiť požiadavky vzdialeného procesora z. V súvislosti s NUMA architektúrou sa hovorí o tzv. NUMA faktore, čo je rozdiel v rýchlosti prístupu k lokálnym a vzdialeným dátam (pamäti).

V najjednoduchšej podobe architektúry NUMA, procesor môže mať lokálnu pamäť, na ktorú dokáže pristupovať rýchlejšie ako lokálnu pamäť iného procesora. Rozdiel v cene za prístup v tomto prípade nie je vysoký, teda NUMA faktor je nízky. Ďalším krokom je zapojenie viacerých procesorov. Pre bežný spotrebiteľský hardware znamená zapojenie viacerých procesorov použitie northbridgu. Problémy s tým spojené sme už popísali v predchádzajúcej sekcii. NUMA je ale tiež, a to predovšetkým, používaná vo veľkých počítačoch, ktoré môžu namiesto northbridgu použiť špecializovaný hardware. Pokiaľ ale nemajú použité pamäťové čipy viacero portov (tzn. že môžu byť použité viacerými zbernicami) stále nám ostáva v systéme to isté úzke miesto. Pamäte s viacerými portami sú drahé a ich podpora je zložitá. Preto sa používajú veľmi zriedkavo. Ďalším krokom sa dostávame k modelu, kde prepojovací mechanizmus poskytuje prístup procesorom, ktoré nie sú priamo pripojené k pamäti (príkladom tohto modelu je Hypertransport od AMD). Veľkosť štruktúr, ktoré je možné týmto spôsobom formovať je obmedzená, pokiaľ nechceme zväčšovať diameter, t.j. maximálnu vzdialenosť medzi ľubovoľnými dvoma uzlami.



Obr. 2.8: Hyperkocky, efektívna topológia zapojenia uzlov NUMA architektúry.

Efektívna topológia pre uzly je hyperkocka, ktorá obmedzuje počet uzlov na 2^C, kde C je počet prepojovacích rozhraní každého uzla. Hyperkocky majú najmenší diameter pre všetky systémy s 2^N procesormi. Obrázok Obr. 2.8 zobrazuje prvé tri hyperkocky. Každá hyperkocka má diameter *C*,

ktorý je absolútne možné minimum. Prvá generácia procesorov AMD založených na tejto technológií (Opteron) mala tri prepojenia (linky) na procesor. Aspoň jeden z procesorov musí mať na jedno prepojenie zapojený southbridge. To znamená, že súčasne je možné efektívne implementovať hyperkocku s $C = 2$. Ďalšia generácia týchto procesorov by mala mať štyri prepojenia na procesor, čím bude možné implementovať hyperkocku s $C = 3$.

To ale neznamená, že nie je možné podporovať väčšie množstvá procesorov. Existujú spoločnosti, ktoré vyvinuli špecializovaný hardware (crossbary), ktorý umožňuje použiť väčšie množstvá procesorov (napr. Horus od firmy Newisys). Tieto crossbary ale zvyšujú NUMA faktor a prestávajú byť efektívne od určitého počtu procesorov. Ďalším krokom by mohlo byť spojenie skupín procesorov a implementácia zdieľanej pamäte pre tieto skupiny. Všetky takéto systémy ale potrebujú špecializovaný hardware a v žiadnom ohľade nespadajú pod bežný spotrebiteľský hardware. Patrí sem napr. IBM x445. Tieto počítače sa predávajú samostatne a je možné vzájomne prepojiť dva alebo štyri z nich tak, aby pracovali ako jeden počítač so zdieľanou pamäťou. Toto prepojenie však predstavuje značný NUMA faktor, s ktorým musí počítať operačný systém aj vyvinuté programy.

Na opačnom konci spektra sú počítače ako napr. Altix od SGI, ktoré boli navrhnuté špecificky pre to, aby sa mohli prepojiť. Použité prepojenie NUMAlink je veľmi rýchle a má nízku latenciu. Tento systém má relatívne nízky NUMA faktor, ale pri počte procesorov, ktoré môže tento systém mať (až tisíce), a obmedzenej kapacite spojení, je NUMA faktor skôr pohyblivý ako statický a môže dosiahnuť neprijateľných úrovní pri veľkom zaťažení.

Oveľa častejšie sa používajú clustre bežných spotrebiteľských počítačov prepojené pomocou vysokorýchlostného sieťového pripojenia. Ale to už nie je NUMA architektúra, neimplementuje zdieľaný adresový priestor a preto nespadá do témy tejto práce.

2.4 Zhrnutie

V tejto kapitole sme nahliadli do kľúčových konceptov moderných pamäťových systémov a videli sme, ako pamäťová hierarchia dodržuje princíp lokality referencií. Popísali sme ako sú organizované rôzne vrstvy pamäťovej hierarchie a videli sme, že rôzne parametre, ako napr. veľkosť bloku, asociativita a stratégia výmeny, ovplyvňujú ako sa presúvajú dáta medzi jednotlivými vrstvami pamäťovej hierarchie a ako nastávajú rôzne typy výpadkov cache. Rozobrali sme pohľad na systém virtuálnej pamäte ako na hierarchiu paralelnú k pamäťovému systému. Vieme, čo je pamäťová latencia a ako môže ovplyvniť čas behu programov. Na praktickom príklade sme ilustrovali čo sa môže stať, keď sa pri návrhu algoritmu neberie v úvahu princíp lokality referencií a pamäťová latencia. Nahliadli sme ako vyzerajú a fungujú SMP a NUMA systémy a s akými faktormi sa tieto systémy musia potýkať.

Čerpali sme z publikácií [6] a [27], v ktorých sa prípadní záujemci môžu dočítať viac.

Kapitola 3

Prehľad pamäťových modelov

V Kapitole 2 sme videli ako veľmi ovplyvňuje spôsob prístupu do pamäte čas behu algoritmu v praxi. Pretože tradičná analýza zložitosti v modeli RAM nevie zachytiť tieto prístupové vzory, potrebujeme mať k dispozícií výpočtové modely, ktoré to dokážu.

V tejto kapitole preskúmame tri výpočtové modely moderných pamäťových systémov, konkrétne external-memory model, hierarchický pamäťový model a ideal-cache model.

External-memory model je popísaný v Sekcii 3.1 a je to model vhodný pre návrh a analýzu algoritmov v externej pamäti. Vzhľadom k určitým nedostatkom je nevhodný pre analýzu cache oblivious algoritmov.

Aj keď je hierarchický pamäťový model veľmi málo používaný, začlenili sme jeho popis do Sekcie 3.2 ako príklad toho, ako boli viacvrstvové pamäťové systémy modelované pred zavedením ideal-cache modelu.

V Sekcii 3.3 popisujeme ideal-cache model. Hoci sa tento model na prvý pohľad môže zdať príliš jednoduchý v porovnaní s reálnymi pamäťovými systémami, je teoreticky aj prakticky opodstatnený, čo dokazujeme v Sekcii 3.4.

3.1 I/O Model (External Memory Model)

Myšlienka analýzy algoritmov počítaním prístupov do pamäte, ktoré algoritmus vykoná, nie je vôbec nová. V oblasti algoritmov a dátových štruktúr vonkajšej pamäte je analýza zložitosti zahŕňajúca počítanie prístupov do pamäte známa už veľa rokov.

Algoritmy a dátové štruktúry vo vonkajšej pamäti pracujú s množinami dát, ktoré veľkosťou ďaleko presahujú hlavnú pamäť. Hlavná myšlienka je, že explicitným riadením pohybu dát medzi vnútornou a vonkajšou pamäťou môžeme minimalizovať dopad latencie spojený s prístupom na disk. Ako také, algoritmy a dátové štruktúry vo vonkajšej pamäti obchádzajú systém virtuálnej pamäte. Dôsledkom explicitného riadenia dát je, že cachovacie parametre vnútornej pamäte môžu byť optimalizované pre špecifický algoritmus, napr. stratégia výmeny môže byť „ušitá“ na mieru, alebo dokonca zmenená počas vykonávania programu.

Pre uľahčenie analýzy algoritmov vo vonkajšej pamäti sa v priebehu minulosti vyvinulo

niekoľko rôznych modelov. Najznámejší z nich je external-memory model, ktorý navrhli Aggarwal a Vitter [1].

Prístup autorov k problému bol skúmať hlavné obmedzenia z hľadiska počtu I/O operácií pre algoritmy vo vonkajšej pamäti. Model, ktorý navrhli pre výskum, pracuje s dvomi pamäťovými vrstvami – vnútornou a vonkajšou. Zložitosť algoritmu sa v tomto modeli meria počtom prístupov do vonkajšej pamäťovej vrstvy (inak povedané počtom I/O operácií) a nazývame ju I/O-zložitosť. Pretože vonkajšia vrstva externého pamäťového modelu je vo väčšine prípadov disk, označuje sa tento model niekedy aj ako I/O model. I/O model je zobrazený na obrázku Obr. 3.1. Parametre modelu sú:

- N označuje počet prvkov množiny vstupných dát algoritmu,
- M označuje počet prvkov, ktoré sa zmestia do vnútornej pamäte,
- B označuje, aký počet prvkov môže byť prenesený v jednom bloku,
- P označuje, aký počet blokov môže byť prenesený súčasne,

kde $1 \leq B \leq M < N$ a $1 \leq P \leq \lfloor M/B \rfloor$.

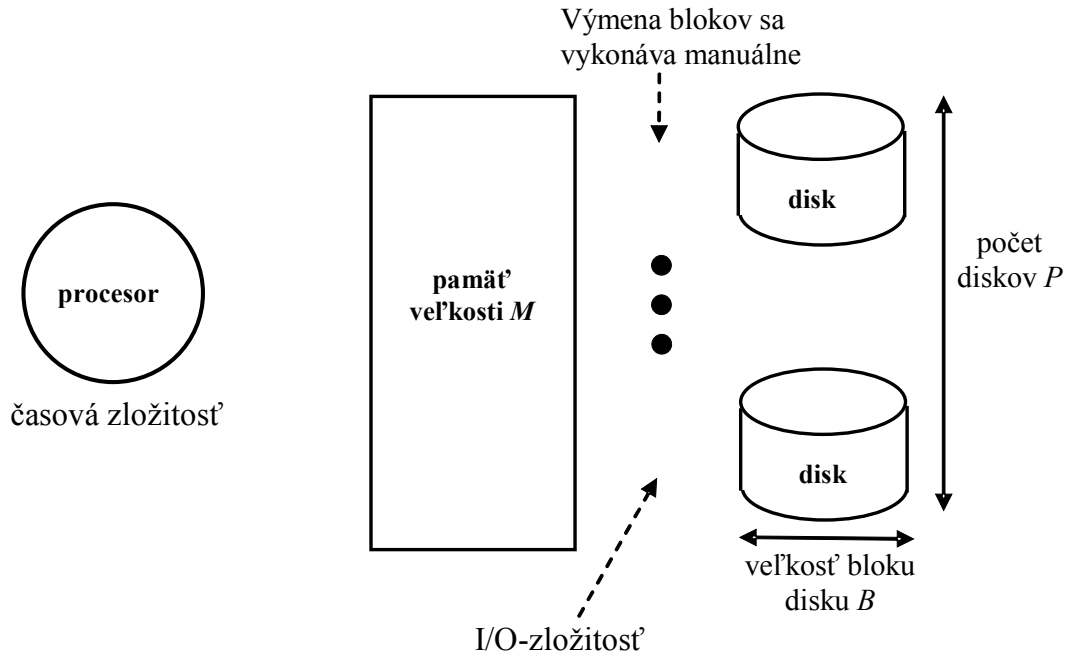
Parametre N, M a B sa označujú tiež ako veľkosť súboru, veľkosť pamäte a veľkosť bloku. Každý prenos bloku má prístup k ľubovolnej súvislej skupine B záznamov na disku. Paralelizmus je prítomný dvoma spôsobmi:

- Každý blok môže naraz preniesť B záznamov (čo modeluje fakt, že disk vie preniesť I/O operáciami blok dát zhruba rovnako rýchlo ako vie preniesť jeden bit).
- Súčasne môže prebiehať P prenosov blokov, čo čiastočne modeluje špeciálne vlastnosti, ktoré môže mať disk (napr. viacnásobné I/O kanály a čítacie/zapisovacie hlavy, schopnosť pristupovať k záznamom, ktoré nie sú uložené súvisle za sebou, v jednej I/O operácií). Parameter P tiež môže modelovať viacero diskov, ktoré môžu súčasne prenášať dáta.

Vyššie spomenutá nerovnosť $1 \leq B \leq M < N$ hovorí, že blok obsahuje aspoň jeden prvok a veľkosť problému (teda vstupných dát) musí presahovať veľkosť vnútornej pamäte. Nerovnosť $1 \leq P \leq \lfloor M/B \rfloor$ hovorí, že počet blokov, ktoré môžu byť prenesené naraz, nesmie presiahnuť počet blokov v pamäti. Je jasné, že nemá zmysel súčasne prenášať viac blokov ako máme miesta vo vnútornej pamäti.

Pri algoritmoch vo vonkajšej pamäti nás často zaujíma okrem I/O-zložitosti aj časová zložitosť algoritmu, pretože manuálne spravovanie dát má väčšinou za následok, že algoritmy navrhuté pre vonkajšiu pamäť sú oveľa zložitejšie ako ich bežné proťajšky. Pre časovú zložitosť algoritmov vo vonkajšej pamäti sa používa RAM model.

Nevýhodou I/O modelu je, že aj keď znalosť presných hodnôt B a M nie je potrebná pri navrhovaní algoritmu v modeli, tieto hodnoty musia byť známe pri implementácií algoritmu. Táto nevýhoda je zrejmá, nakoľko celá myšlienka algoritmov v externej pamäti je vyladiť algoritmus na vlastnosti daného systému manuálnym presúvaním dát. Presné hodnoty B a M sa môžu líšiť v závislosti od systému a tým obmedzujú efektívne použitie implementácie algoritmu iba na niektoré pamäťové systémy. Portovanie takýchto algoritmov na iné platformy môže byť preto netriviálna úloha.



Obr. 3.1: I/O model je dvojvrstvový pamäťový model pozostávajúci z vnútornej a vonkajšej pamäte. Stratégia výmeny blokov sa vykonáva manuálne, t.j. algoritmus sám určuje stratégiu výmeny a preto je znalosť B a M dôležitá.

V pamäťových vrstvách najbližšie k procesoru, tzn. L1 a L2 cache, je stratégia výmeny implementovaná hardwarovo, takže presuny dát nie je možné spravovať manuálne. To ale neznamená, že I/O model nie je aplikovateľný aj na tieto vrstvy pamäte. Keď si budeme vedomí parametrov B, M a stratégie výmeny, tak stále môžeme algoritmus vyladiť. Je ale zrejmé, že optimalizovať algoritmus na niekoľko vrstiev pamäte s tým, že v každej vrstve sa parametre menia, môže byť značne obtiažne.

3.2 Hierarchický pamäťový model

Na rozdiel od dvojúrovňového modelu externej pamäte, hierarchický pamäťový model [13] uvažuje niekoľkoúrovňovú pamäťovú hierarchiu.

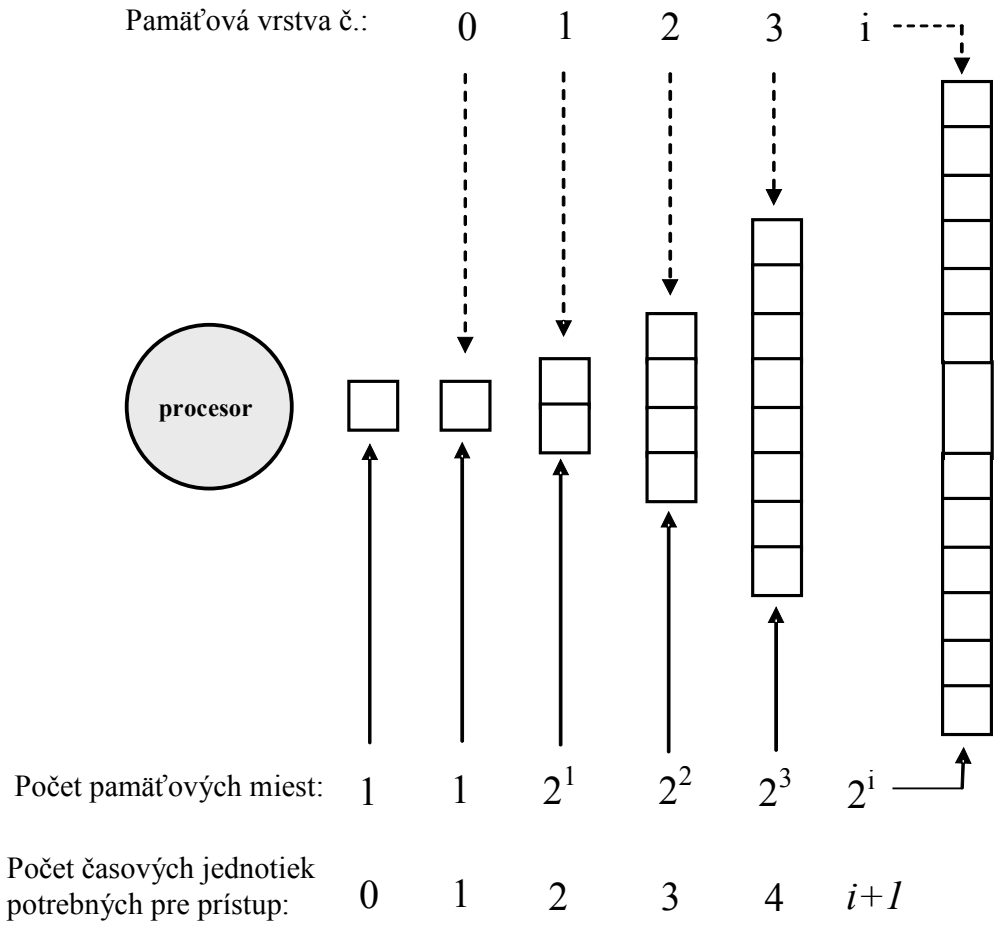
Majme potenciálne neobmedzený počet pamäťových registrov $R_1, R_2, R_3, \ldots$, každý o veľkosti jedného integeru (alebo nejakého iného prvku špecifického pre doménu daného problému). Majme podobné operácie ako pri modeli RAM s tým rozdielom, že prístup do registru R_i trvá $\lceil \log_2(i) \rceil$ a vykonanie akejkoľvek operácie trvá jednu časovú iednotku. Nech sú povolené n-árne operácie. Čas potrebný pre vykonanie n-árnej operácie $R_i \leftarrow f(R_{i_1}, R_{i_2}, \ldots)$ je

$$1 + \lceil \log_2(i) \rceil + \lceil \log_2(i_1) \rceil + \lceil \log_2(i_2) \rceil + \ldots .$$

Na takýto pamäťový model sa môžeme dívať ako na hierarchický pamäťový model pozostávajúci z hierarchie pamäťových vrstiev:

1 pamäťové miesto vyžadujúce nula časových jednotiek pre prístup
1 pamäťové miesto vyžadujúce jednu časovú jednotku pre prístup (úroveň 0)

2 pamäťové miesta vyžadujúce dve časové jednotky pre prístup (úroveň 1)
… atď …
2^i pamäťových miest vyžadujúcich $i + 1$ časových jednotiek pre prístup (úroveň i)



Obr. 3.2: Pre funkciu $f(i) = \lceil \log_2(i) \rceil$ hierarchický pamäťový model pozostáva z niekoľkých pamäťových vrstiev, zväčšujúcich sa násobkom dvoch.

Hierarchický pamäťový model pripomína model RAM. Poskytuje rovnaké operácie a tiež predpokladá potenciálne neobmedzený počet pamäťových registrov $R_1, ..., R_n$ (každý o veľkosti jedného integeru). Tak ako pri modeli RAM, zaujíma nás hlavne čas behu algoritmu, ale na rozdiel od modelu RAM, prístup na miesto R_i trvá $f(i)$ (oproti konštantnému času modelu RAM).
Predpokladá sa, že funkcia $f(i)$ je monotónna neklesajúca a výber f určuje veľkosť a počet pamäťových vrstiev. Spravidla sa volia funkcie napr. $f(i) = x^{\alpha}$, kde $\alpha > 0$ alebo $f(i) = \lceil \log_2(i) \rceil$. Uvedené funkcie f sú iba príklady a je možné aplikovať aj iné funkcie f. Napríklad keď zadefinujme funkciu $f(i)$ ako $f(i) = 1$, ak $i > m$, $f(i) = 0$ inak, tak v podstate meriame zložitosť I/O operácií na počítači s pamäťou veľkosti m (prístup na ktorékoľek pamäťové miesto m je „zadarmo“, každý iný prístup má cenu jedna). Príklad hierarchického pamäťového modelu pre $f(i) = \lceil \log_2(i) \rceil$ je zobrazený na obrázku Obr. 3.2.

Je zrejmé, že hierarchický pamäťový model simuluje správanie pamäťovej hierarchie, pozostávajúcej z čím ďalej tým väčších a pomalších pamäťových vrstiev. Tento model má však svoje nevýhody. Predovšetkým zlyháva v simulácií rôznych stupňov asociativity medzi pamäťovými vrstvami. Ďalej predpokladá, že dáta sú medzi vrstvami presúvané programátorom. V praxi však programátor nemá takúto kontrolu nad dátami hlavne v rýchlejších vrstvách. A na záver, vybrať funkciu f takú, aby odrážala aspoň čiastočne realitu, je veľmi ťažká úloha. Vyžaduje presné znalosti vlastností daného pamäťového systému. Preto je ťažké dosiahnuť, aby čas behu programu predpovedaný modelom odpovedal skutočnému času behu programu v praxi. Funkcia f by musela byť vysoko závislá na špecifických vlastnostiach daného pamäťového systému, pre ktorý bol algoritmus navrhnutý [21]. Hierarchický pamäťový model bol neskôr Aggarwalom a spol. rozšírený o blokový presun medzi pamäťovými vrstvami [14].

3.3 Ideal-cache model

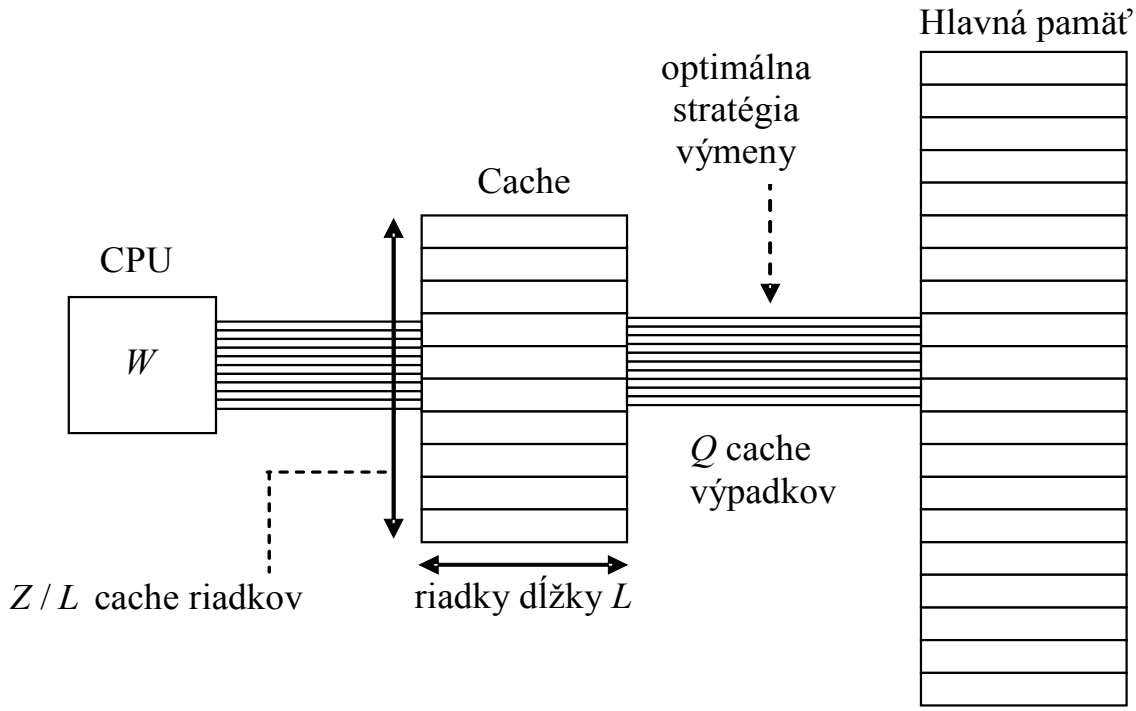
Frigo a spol. [4] navrhli (Z, L) ideal-cache model k štúdiu cache zložitosti algoritmov. Tento model je zobrazený na obrázku Obr. 3.3 a pozostáva z počítača s dvojúrovňovou pamäťovou hierarchiou, ktorá sa skladá z ideálnej (dátovej) cache pamäte o Z slovách a z ľubovoľne veľkej hlavnej pamäte. Pretože skutočná veľkosť slov v počítači je väčšinou malá a pevne daná (napr. 4 byty, 8 bytov a pod.), môžeme predpokladať, že veľkosť slova je konštanta. Cache pamäť je rozdelená do cache riadkov, každý pozostávajúci z L po sebe nasledujúcich slov, ktoré sa medzi cache pamäťou a hlavnou pamäťou prenášajú vždy naraz. Návrhári cache pamätí väčšinou používajú $L > 1$ za účelom vylepšenia spatial locality k umoreniu režijných nákladov spojených s presúvaním cache riadkov. Obvykle sa predpokladá, že cache pamäť je „vysoká“, teda že platí $Z = \Omega\left(L^2\right)$, čo je v praxi väčšinou pravda.

Procesor môže adresovať len slová, ktoré sa nachádzajú v cache pamäti. Ak odkazované slovo patrí riadku, ktorý už je v cache pamäti, nastáva cache hit a slovo sa odovzdá procesoru. V opačnom prípade nastáva cache výpadok a riadok sa musí preniesť do cache pamäte. Ideálna cache pamäť je plne asociatívna, tzn. že riadky môžu byť uložené kdekoľvek v cache pamäti. V prípade, ak je cache pamäť plná, musí sa nejaký cache riadok vyhodiť. Ideálna cache pamäť používa optimálnu off-line stratégiu výmeny cache riadka, na ktorý sa pristupuje čo najďalej v budúcnosti a teda dobre využíva princíp lokality referencií (pozn.: pod pojmom off-line stratégia výmeny rozumieme takú stratégiu výmeny, ktorá ma úplnú znalosť budúcej postupnosti prístupov do pamäte). Presun dát medzi vrstvami sa deje automaticky.

Algoritmus so vstupnými dátami veľkosti n je skúmaný z hľadiska časovej zložitosti $W(n)$ - čas behu programu v modeli RAM a cache zložitosti $Q(n; Z, L)$ - počet cache výpadkov spôsobených algoritmom ako funkcia veľkosti Z a dĺžky riadku L ideálnej cache pamäte. Ak sú hodnoty premenných Z a L jasné z kontextu, zapisuje sa cache zložitosť jednoducho ako $Q(n)$.

Hovoríme, že algoritmus je „cache aware“, ak obsahuje parametre (nastavované pri kompilácií alebo spúšťaní programu), ktorých nastavovaním môžeme optimalizovať cache zložitosť pre konkrétnu veľkosť cache pamäte a dĺžku riadku. V opačnom prípade hovoríme, že algoritmus je „cache oblivious“. Podľa tejto definície sú všetky algoritmy, ktoré nevenujú špeciálnu pozornosť pamäťovému systému, cache-oblivious. Teda všetky tradičné algoritmy navrhnuté v RAM modeli sú cache-oblivious. Preto má zmysel rozlišovať medzi cache-oblivious algoritmami a optimálnymi

cache-oblivious algoritmami, t.j. cache-oblivious algoritmami, ktoré spôsobujú asymptoticky minimálny počet cache výpadkov. V kontexte cache-oblivious algoritmov budeme pod pojmom cache rozumieť nielen L1, L2, ... cache pamäte ale aj menšiu z ktorýchkoľvek dvoch po sebe nasledujúcich pamäťových vrstiev.



Obr. 3.3: Ideal-cache model je dvojvrstvový pamäťový model. Cache pamäť je plne asociatívna a dáta sa presúvajú medzi hlavnou pamäťou a cache pamäťou automaticky v súlade s optimálnou stratégiou výmeny. Predpokladá sa, že cache pamäť je „vysoká“, t.j. spĺňa podmienku $Z = \Omega\left(L^2\right)$.

Aby sme získali predstavu o pojme cache-aware algoritmu, vezmime si úlohu násobenia dvoch $n \times n$ matíc A a B, výsledkom čoho dostaneme $n \times n$ maticu C. Predpokladáme, že všetky tri matice sú uložené po riadkoch. Za účelom zjednodušenia analýzy ďalej predpokladáme, že n je dostatočne veľké, tzn. $n > L$. Obvyklý spôsob ako násobiť matice na počítači s cache pamäťou, je použiť blokový algortimus. Myšlienka spočíva v tom, pozerať sa na každú maticu M ako na zostavu $(n/s) \times (n/s)$ podmatíc M_{ij} (bloky), každá veľkosti $s \times s$, kde s je ladiaci parameter:

```
BLOCK-MULT(A,B,C,n)
for i ← 1 to n/s
  do for j ← 1 to n/s
     do for k ← 1 to n/s
        do ORD-MULT(A_ik, B_kj, C_ij, s)
```

Podprogram `ORD-MULT` spočíta $C \leftarrow C + AB$ na maticiach veľkosti $s \times s$ použitím obyčajného $O\left(s^3\right)$ algoritmu. Algoritmus popisovaný v príklade pre jednoduchosť predpokladá, že s je

celočíselným deliteľom n. V skutočnosti s a n nemusia byť v žiadnom (špeciálnom) vzťahu, čo sa odrazí iba v zložitejšom zápise kódu.

V závislosti na veľkosti cache pamäte na počítači, na ktorom bude `BLOCK-MULT` spustený, môžeme zmenou parametra s optimalizovať algoritmus tak, aby bežal rýchlejšie. Teda program `BLOCK-MULT` je cache-aware algoritmus. Aby sme minimalizovali cache zložitosť, zvolíme hodnotu s najväčšiu možnú tak, aby sa tri $s \times s$ podmatice súčasne zmestili do cache pamäte. Matica veľkosti $s \times s$ je uložená v $\Theta\left(s + s^2 / L\right)$ cache riadkoch. Z predpokladu, že cache pamäť je „vysoká“ vidíme, že $s = \Theta\left(\sqrt{Z}\right)$. Teda každé zavolanie `ORD-MULT` spôsobí najviac $Z / L = \Theta\left(s^2 / L\right)$ cache výpadkov potrebných k presunutiu matíc do cache pamäte. Z toho vyplýva, že cache zložitosť celého algoritmu je

$$\Theta\left(1 + n^2 / L + \left(n / \sqrt{Z}\right)^3 \left(Z / L\right)\right) = \Theta\left(1 + n^2 / L + n^3 / L\sqrt{Z}\right),$$

nakoľko algoritmus musí prečítať n^2 prvkov umiestnených v n^2 / L cache riadkoch. Také isté medze môžeme dosiahnuť použitím jednoduchého cache-oblivious algoritmu [4, s.286], ktorý nevyžaduje žiadne ladiace parametre, ako bol parameter s v prípade `BLOCK-MULT`.

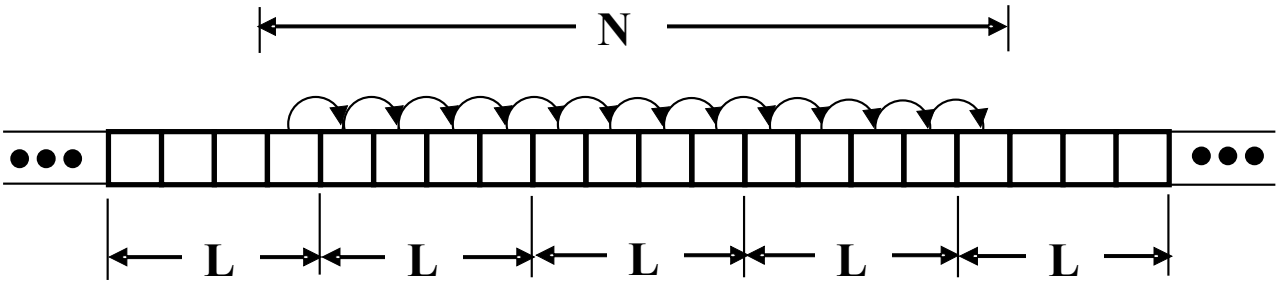
Na prvý pohľad sa môže zdať, že neznalosť parametrov Z a L prestavuje značné obmedzenie. Ukážeme si, že to nie je pravda a že neznalosť parametrov Z a L má prekvapivo silné dôsledky. Prvým dôsledkom cache-oblivious prístupu je, že ak cache-oblivious algoritmus dokáže používať dve pamäťové vrstvy optimálne, potom musí automaticky byť schopný používať *akékoľvek* dve pamäťové vrstvy optimálne. To znamená, že cache-oblivious algoritmy, ktoré sa správajú optimálne v dvojúrovňovej pamäťovej hierarchií sa budú správať optimálne vo všetkých vrstvách pamäťovej hierarchie s viac ako dvoma vrstvami. Druhým dôsledkom cache-oblivious prístupu je, že algoritmus, ktorý beží dobre na jednom počítači, by mal bežať rovnako dobre aj na ktoromkoľvek inom počítači – vlastnosť, ktorá úplne odstraňuje problémy s portovaním algoritmov na rôzne platformy.

Na druhej strane ideal-cache model pracuje s určitými predpokladmi, ktoré sa môžu zdať nerealistické v porovnaní s modernými cache pamäťmi a pamäťovými systémami. Za prvé, optimálna stratégia výmeny je nerealistická, nakoľko vyžaduje znalosť behu programu v budúcnosti. Za druhé, veľmi málo moderných pamäťových systémov má iba dve úrovne pamäte. Za tretie, plná asociativita je veľmi zriedkavá, drvivá väčšina cache pamätí má obmedzenú asociativitu. Overenie správnosti predpokladov ideal-cache modelu nájdeme v Sekcii 3.4.

3.3.1 Techniky návrhu cache-oblivious algoritmov

Podľa Demaina [28] sú dve hlavné techniky návrhu cache-oblivious algoritmov sekvenčné skenovanie a technika rozďeluj a panuj.

Skenovanie poľa N prvkov v I/O modeli spôsobí načítanie $\lceil N / L \rceil$ blokov z disku do vnútornej pamäte. V ideal-cache modeli rovnaká úloha spôsobí nanajvýš $\lceil N / L \rceil + 1$ cache výpadkov. Tento rozdiel nastáva z dôvodu, že pre ideal-cache model nepoznáme parametre Z a L. Dôsledkom toho nemôžeme v pamäti zarovnať pole s hranicami cache riadkov, viď. obrázok Obr.3.4. Z tohto príkladu vyplýva, že prístup na súvislé miesto v pamäti o veľkosti jedného bloku znamená v najhoršom prípade prístup na dva fyzické bloky.



Obr. 3.4: Dôsledkom zlého zarovnania spôsobí cache-oblivious skenovanie poľa N prvkov v najhoršom prípade $\lceil N/L \rceil + 1$ cache výpadkov.

Technika návrhu algoritmov rozdeľuj a panuj opakovane zjemňuje veľkosť riešeného problému až kým nedosiahne nejaký základný prípad, ktorý je už ľahko riešiteľný. Napríklad obyčajný mergesort opakovane delí pole prvkov na dve čiastkové polia polovičnej veľkosti. Základný prípad nastáva, keď čiastkové polia obsahujú iba jeden prvok a týmpádom sa dajú ľahko zlúčiť.

V kontexte cache-oblivious algoritmov technika rozdeľuj a panuj tiež znamená rozdelenie problému na menšie časti. Analogicky k príkladu s mergesort algoritmom, základný prípad nastáva, keď sa podproblém stane ľahko riešiteľným – kým pre tradičné algoritmy (t.j. v modeli RAM) to znamená riešiteľným konštantným počtom inštrukcií, v kontexte cache-oblivious algoritmov to znamená, že riešenie nespôsobí žiadne ďalšie výpadky cache. Spravidla to nastáva, keď sa podproblém zmestí do cache pamäte (pozn.: To neznamená, že algoritmus prestane ďalej deliť podproblém v momente keď sa zmestí do cache, nakoľko to by z algoritmu robilo cache-aware algoritmus. Znamená to len toľko, že tento moment v rekurzívnom delení problému poskytuje základný prípad pre analýzu cache zložitosti.).

3.4 Overenie správnosti predpokladov ideal-cache modelu

Ideálna cache pamäť je abstrakciou reálnych pamäťových systémov. Keď ju porovnáme s charakteristikou moderných pamäťových systémov, napr. tých popísaných v Kapitole 2, jasne vidíme, že charakteristika ideálnej cache pamäte činí z ideal-cache modelu zjednodušenie reálnych pamäťových systémov.

Počet cache výpadkov, ktoré spôsobí algoritmus v ideal-cache modeli, je vrámci určitého konštantného faktora počet cache výpadkov, ktoré algoritmus spôsobí v reálnom pamäťovom systéme [4]. Toto je kľúčová vlastnosť tohto modelu.

Ak chceme byť schopný čo najpresnejšie aproximovať cache zložitosť v reálnych pamäťových systémoch, musíme mať dobrý prehľad o tom, v čom sa od nich ideálna cache pamäť líši a ako tieto rozdiely ovplyvňujú cache zložitosť.

Z teoretického hľadiska je správnosť predpokladov ideal-cache modelu overená Frigom a spol v [4]. Ukázali, že pomocou určitých obmedzení je možné pozmeniť ideal-cache model na realistický cache model s LRU stratégiou výmeny, viacerými vrstvami pamäte a s priamym mapovaním. V tejto práci nebudeme formálne dokazovať túto transformáciu (záujemci nájdu formálny postup v [4]), ale neformálne sa pozrieme na jednotlivé obmedzenia a na to, ako slabý alebo silný je ideal-cache model v porovnaní s reálnymi pamäťovými systémami.

3.4.1 Predpoklad: Optimálna stratégia výmeny

Frigo a spol. [4] vo svojej práci používajú výsledky práce Sleatora a Tarjana [22], ktorí sa zaoberali efektívnosťou rôznych on-line stratégií výmeny, ako napr. LRU a FIFO, v porovnaní s optimálnou off-line stratégiou výmeny. Pod pojmom on-line stratégia výmeny sa myslí taká stratégia výmeny, ktorá nemá žiadne informácie o prístupoch do pamäte v budúcnosti a pod pojmom off-line stratégia výmeny sa myslí taká stratégia výmeny, ktorá ma úplnú znalosť budúcej postupnosti prístupov do pamäte.

Sleator a Tarjan ukázali, že pre akúkoľvek konštantu $c > 1$, algoritmus spôsobí na LRU alebo FIFO cache pamäti veľkosti M maximálne c krát toľko cache výpadkov ako by spôsobil ten istý algoritmus na optimálnej cache pamäti veľkosti $(1 - 1/c)M$. To sa dá vyjadriť ako

$$Q_{LRU}(N, M, B) = cQ_{OTP}(N, (1 - 1/c)M, B),$$

kde Q_{LRU} je počet cache výpadkov s LRU stratégiou výmeny a Q_{OPT} je počet cache výpadkov s optimálnou stratégiou výmeny. Frigo a spol. volili $c = 2$ a môžu preto tvrdiť, že algoritmus spôsobujúci $2Q$ cache výpadkov na LRU-cache pamäti veľkosti M s veľkosťou bloku B spôsobí najviac Q cache výpadkov na optimálnej cache pamäti polovičnej veľkosti s rovnakou veľkosťou bloku. Preto je možné tvrdiť, že stratégia výmeny LRU je rovnako dobrá ako optimálna stratégia výmena až na konštantný faktor cache výpadkov a konštantný faktor prebytočných cache riadkov.

Frigo a spol. ďalej definujú, že cache zložitosť algoritmu je *regular*, ak platí:

$$Q(N, M, B) = 0\,(Q(N, 2M, B)).$$

To znamená, že ak sa zníži počet cache riadkov na polovicu, cache zložitosť algoritmu je ovplyvnená iba o konštantný faktor. Táto podmienka je dôležitá, nakoľko nám zaručuje, že algoritmus nebude pracovať s LRU-cache pamäťou najhorším možným spôsobom.

Čo znamená najhorší možný spôsob práce s LRU-cache pamäťou a ktoré algoritmy majú nepravidlnú cache zložitosť si ukážeme na nasledujúcom príklade. Vezmime si algoritmus, ktorý cyklicky skenuje pole $M/B + 1$ prvkov s predpokladom, že tieto prvky sú uložené v $M/B + 1$ rôznych blokoch. Po prejdení prvých M/B prvkov bude cache pamäť plná, takže prístup na posledný prvok poľa spôsobí výpadok. Stratégia výmeny LRU vyhodí blok obsahujúci prvý prvok poľa. Lenže ďalší prvok poľa, na ktorý sa má pristúpiť, sa nachádza v práve vyhodenom bloku (pripomíname, že algoritmus prechádza pole cyklicky). Takže nastane výpadok cache a musí sa vyhodiť ďalší blok. Stratégia výmeny LRU vyhodí blok obsahujúci druhý prvok poľa, na ktorý sa ale bude vzápätí pristupovať. Tento vzor, kedy každý prístup spôsobí cache výpadok, pokračuje až kým algoritmus nejako neskončí. Cache zložitosť algoritmu závisí od toho, koľko prechodov algoritmus spraví nad vstupným poľom.

Zdvojenie počtu cache riadkov na $2M/B$ by malo za následok, že cache výpadky by

spôsobilo iba prvých $M/B+1$ prístupov do pamäte a všetky následné prístupy do pamäte by už boli na bloky, ktoré sú už v cache pamäti, bez ohľadu na to, koľko cyklov algoritmus spraví nad vstupným poľom. Takže v tomto prípade závisí cache zložitosť algoritmu na parametroch M a B a nie na počte prechodov nad vstupným poľom. Tento popisovaný najhorší možný prípad práce s LRU-cache pamäťou nespĺňa podmienku regularity, takže algoritmy s regulárnou cache zložitosťou sa takto správať nebudú.

Spojením výsledku práce Sleatora a Tarjana s podmienkou regularity dostávame Dôsledok 13 z [4], ktorý hovorí, že každý algoritmus s cache zložitosťou splňujúcou podmienku regularity v ideal-cache modeli s veľkosťou M bude mať asymptoticky rovnakú cache zložitosť v LRU-cache pamäti rovnakej veľkosti.

Frigo a spol. predpokladali reálnu LRU stratégiu výmeny. V reálnych pamäťových systémoch je LRU väčšinou nahradená nejakou aproximáciou alebo sa použije náhodná stratégia výmeny. Takže Dôsledok 13 z [4] nie je celkom pravdivý pre reálne pamäťové systémy. Na druhej strane, ako sme už spomínali v Kapitole 2, voľba medzi LRU a náhodnou stratégiou výmeny má malý dopad na správanie sa cache pamätí v praxi – aspoň pre cache pamäte určitej minimálnej veľkosti. Avšak mali by sme si uvedomiť, že tieto praktické pozorovania iba vyjadrujú podobné správanie týchto dvoch stratégií výmeny v priemernom prípade. Je možné, že tým, ako stratégia LRU dodržuje princíp lokality referencií, má v kontexte cache-oblivious algoritmov navrch.

3.4.2 Predpoklad: Dve pamäťové vrstvy

Reálne pamäťové systémy majú väčšinou 3 až 5 pamäťových vrstiev, preto sa musíme uistiť, že predpoklad dvoch pamäťových vrstiev ideal-cache modelu je postačujúci. Inak povedané, má cache-oblivious algoritmus s optimálnou cache zložitosťou v dvojvrstvovom ideal-cache modeli optimálnu cache zložitosť aj vo viacvrstvovej pamäťovej hierarchií so stratégiou výmeny LRU? Na túto otázku sa pokúsime odpovedať dvoma spôsobmi:

1. Za predpokladu, že všetky vrstvy pamäte dodržujú vlastnosť „inclusion property“ (vysvetlenú v sekcii 2.1) a sú spravované optimálnou stratégiou výmeny, je to intuitívne pravda. Pripomeňme si, že vlastnosť inclusion property znamená, že dáta sa nemôžu nachádzať vo vrstve i ak sa nenachádzajú aj vo vrstve $i+1$ (kde vrstva i je bližšie k procesoru ako vrstva $i+1$). Optimálna stratégia výmeny z definície zaručí minimálny počet cache výpadkov v každej vrstve. V dôsledku toho sa celá hierarchia pamäťových vrstiev využíva optimálne. Ak teraz aplikujeme Dôsledok 13 z [4] na každú vrstvu pamäťovej hierarchie, zmeníme optimálnu stratégiu výmeny na LRU. Toto obmedzenie zvýši cache zložitosť každej vrstvy o nejaký konštantný faktor v závislosti na použitom algoritme.

   Frigo a spol. [4] dokazujú asymptotickú optimálnosť viacvrstvovej LRU pamäťovej hierarchie viac formálne. Najskôr dokazujú, že použitím LRU stratégie výmeny vo viacvrstvových hierarchiách sa zachová vlastnosť inclusion property. Tento dôkaz je dôležitý, pretože keby sa vlastnosť inclusion property nezachovala naprieč všetkými vrstvami LRU pamäťovej hierarchie, potom jediným miestom, kde by sa zachoval princíp lokality referencií, by bola pamäťová vrstva najbližsie k procesoru.

2. Frigo a spol. tvrdia, že v ktorejkoľvek vrstve i v hierarchií viacerých pamäťových vrstiev so stratégiou výmeny LRU, výpadky cache dejúce sa v nižších vrstvách nie sú vidieť. Vrstva i sa

chová presne tak, ako keby bola prvou z vrstiev v dvojvrstvovej hierarchii. To znamená, že pre akúkoľvek postupnosť prístupov do pamäte obsahuje presne tie isté dáta ako by obsahovala, keby bola prvou vrstvou v dvojvrstvovej hierarchii vykonávajúcej rovnakú postupnosť prístupov do pamäte. V dôsledku toho môžeme analyzovať správanie pamäťového systému na rozhraní ktorýchkoľvek dvoch, po sebe idúcich, cache pamätí v ideal-cache modeli.

Takže ideal-cache model je aplikovateľný na ktorékoľvek rozhranie dvoch pamäťových vrstiev v hierarchickom pamäťovom systéme so stratégiou výmeny LRU. A na základe Dôsledku 13 z [4], každú vrstvu môžeme upraviť tak, aby používala stratégiu LRU za cenu zvýšenia cache zložitosti každej vrstvy o konštantný faktor.

Tak ako v prípade redukcie optimálnej stratégie výmeny na LRU, odstránenie predpokladu dvoch pamäťových vrstiev zvyšuje cache zložitosť v každej pamäťovej vrstve o konštantný faktor.

3.4.3 Predpoklad: Plná asociativita a automatický presun dát

Ideálna cache pamäť predpokladá, že cache bloky sa nahradzujú v prípade cache výpadku automaticky. Z pohľadu programátora je to v podstate korektný predpoklad. Medzi menšími pamäťovými vrstvami sa výmena blokov deje automaticky hardwarovo a o výmenu medzi hlavnou pamäťou a diskom sa stará operačný systém. Takže programátor sa nemusí vôbec starať o výmenu blokov.

Ďalej ideálna cache pamäť predpokladá plnú asociativitu aj keď sú pamäťové vrstvy bližšie k procesoru väčšinou 2, 4, až 8-cestné množinovo asociatívne. Otázkou ostáva ako efektívne sú operačné systémy schopné implementovať automatickú výmenu blokov a plnú asociativitu?

Podľa Friga a spol. [4] je možné udržovať plne asociatívnu LRU cache pamäť vrámci bežnej pamäte tak, že necháme presuny dát na softwari. Použitím hashovacích techník je možné dosiahnuť implementáciu, ktorá poskytuje prístup ku ktorémukoľvek cache riadku s očakávaným časom $0(1)$, za použitia $0(M/B)$ záznamov, každý veľkosti $0(B)$. V nasledujúcich odstavcoch naznačíme ako je to možné dosiahnuť.

Uvážme problém rovnomerného rozdelenia ľubovoľnej podmnožiny S úplne (tzn. lineárne) usporiadaného univerza prvkov U do množiny priehradok L, kde $L << U$. Použitím 2-univerzálnych hashovacích funkcií [23] môžeme zabezpečiť, že bez ohľadu na to, ktoré dva rozdielne prvky namapujeme z U do L, pravdepodobnosť, že skončia v rovnakej priehradke, ostane rovnaká. Vďaka tomu máme zaručené, že bez ohľadu na to, akú podmnožinu $S \subset U$ zvolíme, bude rovnomerne rozdelená medzi priehradky L.

Analogicky pre náš problém: Nech M_{big} a M_{small} označujú veľkosti dvoch susediacich pamäťových vrstiev a nech B označuje veľkosť ich blokov. Tieto dve vrstvy obsahujú M_{big}/B a M_{small}/B blokov. Menšia z vrstiev je x-cestná množinovo asociatívna, takže je rozdelená na M_{small}/x priehradok, kde každá priehradka obsahuje x blokov.

Máme: M_{big} je úplne usporiadané univerzum M_{big}/B blokov a M_{small} je množina priehradok. Môžeme preto použiť 2-univerzálne hashovacie funkcie a namapovať ľubovoľnú podmnožinu M_{small}/B blokov väčšej z vrstiev na menšiu vrstvu. Pretože sme použili 2-univerzálne hashovacie funkcie, predpokladáme, že namapovaná podmnožina bude rovnomerne rozdelená medzi M_{small}/x priehradok.

Vrámci každej priehradky môžeme bloky spojiť do obojstranného spojového zoznamu. Vďaka 2-univerzálnym hashovacím funkciám sa v každej priehradke nachádza konštantný počet blokov – presne x blokov, takže vrámci priehradky je možné udržovať LRU stratégiu výmeny v konštantnom čase. Časová zložitosť 2-univerzálnych hashovacích funkcií je tiež konštantná, takže celý proces má konštantnú asymptotickú časovú zložitosť.

3.4.4 Predpoklad: „Vysoká“ cache pamäť

Posledným predpodkladom ideálnej cache pamäte bolo, že je „vysoká“. Tento predpoklad sa dá zapísať ako $Z = \Omega\left(L^2\right)$ a znamená, že počet riadkov v cache pamäti (Z/L) je väčši ako veľkosť bloku L. Spomedzi iných, napr. Prokop [24] používa predpoklad vysokej cache pamäte pri dôkazoch optimálnej cache zložitosti algoritmov násobenia matíc, transpozície matice a FFT. Niekedy stačí slabší predpoklad $Z = \Omega\left(L^{1+\gamma}\right)$, kde $\gamma > 0$. Použili ho napr. Brodal a Fagerberg [11] pri analýze algoritmu lazy funnelsort, ktorý popisujeme v Kapitole 4.

Brodal a Fagerberg [25] zdôrazňujú dôležitosť predpokladu vysokej cache pamäte. Ukazujú, že optimálne triedenie založené na porovnávaní nie je možné dosiahnuť bez tohto predpokladu. Frigo a spol. [4] poukazujú na to, že v praxi tento predpoklad cache pamäte väčšinou splňujú. Keď sa pozrieme na charakteristiky procesorov rodiny Intel (Tab. 2.2 v Kapitole 2) vidíme, že predpoklad vysokej cache pamäte je naozaj realistický.

3.5 Zhrnutie

V tejto kapitole sme popísali I/O model, hierarchický pamäťový model a ideal-cache model. V prvých dvoch modeloch je znalosť charakteristiky pamäťového systému nevyhnutná k návrhu efektívneho algoritmu. V ideal-cache modeli nie je znalosť charakteristiky pamäťového systému potrebná.

Ideálna cache pamäť je teoreticky obhájiteľná. Je možné ju redukovať na cache pamäť, ktorá pripomína reálne pamäťové hierarchie a táto redukcia zvyšuje cache zložitosť algoritmu iba o konštantný faktor. Obhajoba predpokladov ideal-cache modelu nám hovorí, že algoritmus, ktorý je asymptoticky optimálny v ideal-cache modeli, je tiež asymptoticky optimálny v reálnych pamäťových systémoch. Tento vzťah je síce veľmi dôležitý, ale sám o sebe nám nič nehovorí o tom, o koľko presne je cache zložitosť algoritmu horšia v reálnych pamäťových systémoch v porovnaní s ideal-cache modelom. Ako sme videli, faktor oddeľujúci reálnu cache zložitosť od cache zložitosti v ideal-cache modeli závisí na charakteristikách pamäťového systému, ako je stupeň asociativity a stratégia výmeny, a preto nevieme určiť jeho hodnotu.

Kapitola 4

Algoritmy

Pre dané pole N prvkov úloha triedenia pozostáva z preusporiadania prvkov do neklesajúceho poradia. Formálne povedané, zotriediť N prvkov $x_1,...,x_N$, za predpokladu lineárneho usporiadania R, znamená nájsť takú permutáciu π N prvkov $1,...,N$, že $x_{\pi(i)} R x_{\pi(i+1)}$ pre $1 \leq i < N$. Ako bude uvedené neskôr (v Kapitole 5), v tejto práci pre účely testov nepoužívame generátor permutácií. Triedenie dát patrí medzi základné problémy informatiky. Existuje veľa metód, ktoré sa dajú na tento problém úspešne aplikovať, ale v tejto kapitole sa budeme zaoberať iba veľmi malou časťou z nich.

V Sekcii 4.1 sú popísané triediace cache-aware algoritmy navrhnuté LaMarcom a Ladnerom [2]. V Sekcii 4.2 je popísaný triediaci cache-oblivious algoritmus funnelsort a jeho modifikácia lazy funnelsort. V Sekcii 4.3 uvádzame triediaci cache-oblivious algoritmus distribution sort.

4.1 Cache-Aware algoritmy: multimergesort, multiquicksort, memory-optimized heapsort

Na prácu Aggarwalla a Vittera [1] voľne naviazali vo svojej práci LaMarca a Ladner [2]. Podľa LaMarcu a Ladnera sa niektoré algoritmické myšlienky z externého pamäťového modelu môžu dobre uplatniť aj v hierarchickom pamäťovom modeli. Jedná sa hlavne o viaccestné zlievanie („multimerging“) a viaccestné rozdeľovanie („multipartitioning“). LaMarca a Ladner ďalej poukazujú na to, že aj keď existuje veľa podobností medzi I/O modelom a hierarchickým pamäťovým modelom, sú medzi nimi rozdiely. Algoritmus vo vonkajšej pamäti má kontrolu nad tým, ktoré dáta sú v hlavnej pamäti a ktoré na disku. Na druhej strane, keď nastane výpadok pri hardwarových cache pamätiach, algoritmus nemá na výber, kam sa umiestni nový blok alebo ktorý blok sa vyhodí z cache pamäte. Ďalším rozdielom, podľa LaMarcu a Ladnera, je fakt, že jediné čo sa v I/O modeli počíta, sú I/O operácie s diskom. Výpočet je „zadarmo“ z dôvodu, že diskové I/O operácie sú rádovo drahšie ako výpočtový krok. A nakoniec je tu rozdiel vo veľkosti. Cache pamäť, nachádzajúca sa najbližšie k hlavnej pamäti, je relatívne veľká v pomere k hlavnej pamäti oproti I/O modelu, kde kapacita disku je väčšinou pomerovo oveľa väčšia vzhľadom k hlavnej pamäti.

LaMarca a Ladner vo svojej práci použili nasledujúce prístupy k meraniu algoritmov a analýze ich zložitosti [2]. Zamerali sa na tri hlavné ukazatele: počet inštrukcií, počet cache výpadkov a čas behu programu.

Počet inštrukcií skúmaných algoritmov bol meraný použitím nástroja ATOM [15]. ATOM je nástroj vyvinutý pre inštrumentáciu programov na počítačoch Alpha. Použitím ATOMu bol upravený každý algoritmus tak, aby zvýšil počítadlo vždy, keď sa vykoná jedna inštrukcia. Týmto spôsobom je možné spustiť algoritmus a získať počet vykonaných inštrukcií.

Pre účely merania počtu cache výpadkov autori vyvinuli jednoduchú knižnicu, ktorá simuluje správanie priamo mapovanej cache pamäte. Táto simulovaná cache pamäť má funkciu, ktorá pre adresu v pamäti vráti, či prístup na danú adresu spôsobil cache výpadok. Všetky algoritmy boli pomocou ATOMu upravené tak, aby pre každý zápis alebo čítanie použili túto knižnicu.

Posledným ukazateľom je nameraný čas behu algoritmov. Čas behu algoritmov bol meraný pomocou hardwarového počítadla cyklov. Od hodnoty počítadla cyklov po vykonaní algoritmu sa odčíta hodnota počítadla cyklov pred vykonaním algoritmu, čím dostaneme presný počet vykonaných cyklov. Vydelením tohto čísla počtom cyklov za sekundu dostaneme čas behu programu v sekundách. Za priemernú hodnotu času behu programu bol zvolený medián z 15 opakovaní.

Autori neanalyzovali priamo cache zložitosť. Namiesto toho analyzovali prístupy do cache pamäte. Pre analýzu prístupov do cache pamäte zvolili jednoduchý model. Predpokladajú, že existuje veľká pamäť rozdelená na bloky a menšia cache pamäť rozdelená na C blokov. Na vstupe je n kľúčov a B je počet kľúčov, ktoré sa zmestia do jedného cache bloku. Ďalej predpokladajú, že kľúče sú uložené v poli veľkosti n/B súvislých pamäťových blokov. V priamo mapovanej cache pamäti sa každý blok pamäte namapuje presne na jeden blok v cache pamäti. Autori predpodkladajú jednoduché mapovanie, tzn. pamäťový blok x sa namapuje na cache blok $x \bmod C$. V každom okamihu je každý blok y v cache pamäti asociovaný s práve jedným blokom pamäte x takým, že $y = x \bmod C$. Prístup na pamäťový blok x nazveme výpadok, ak sa x nenachádza v cache pamäti. Následkom výpadku sa blok x zanesie do cache pamäte a blok nachádzajúci sa na mieste $x \bmod C$ sa vyhodí. Na skúmaný algoritmus v tomto kontexte autori nahliadajú ako na postupnosť prístupov k blokom v pamäti. Predpokladajú, že na začiatku nie sú v cache pamäti žiadne bloky, na ktoré sa bude počas vykonávania programu pristupovať. Nerozlišuje sa medzi čítaním a zápisom, pretože autori predpokladajú architektúru copy-back s write-bufferom pre cache [16]. Tento jednoduchý model neodráža všetky prístupy do pamäte, ktoré môže skutočný program vykonávať. Napríklad polia nemusia byť nutne alokované v súvislých úsekoch pamäte. Analýza algoritmov preto v niektorých prípadoch uvažuje zjednodušujúce predpoklady.

Algoritmy popisované v tejto sekcii sú cache-aware.

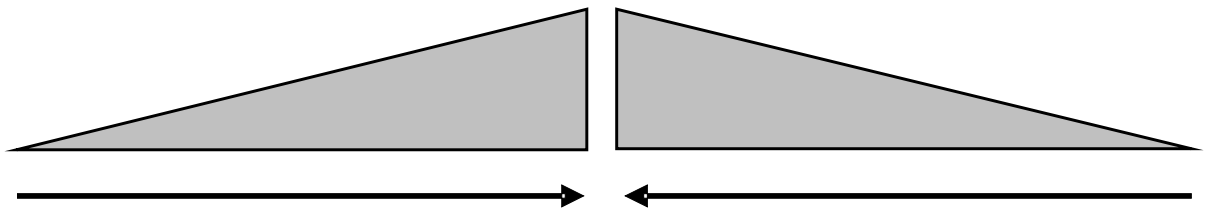
4.1.1 Mergesort, tiled mergesort, multimergesort

Mergesort je triediaci algoritmus založený na princípe postupného rekurzívneho rozdeľovania triedenej postupnosti na úseky. Jeho základná myšlienka je veľmi jednoduchá. Pole obsahujúce triedené čísla rozdelíme na dve polovice a každú z nich samostatne zotriedime. Zo zotriedených čiastkových úsekov potom zostavíme výslednú zotriedenú postupnosť metódou zlučovania. Zotriedenie oboch čiastkových úsekov poľa sú úlohy rovnakého typu ako pôvodná úloha. Vyriešime ich preto rovnakým postupom. Toto rozkladanie na stále menšie úseky pokračuje tak dlho, až kým získame úseky dĺžky jedna, ktoré sú už samy o sebe zotriedené. Zlučovanie dvoch zotriedených úsekov do jedného sa vykoná jednoducho – oba zlučované úseky sa postupne prechádzajú a vždy menšie z čísel sa premiestni do vytváranej výslednej postupnosti (potrebujeme ešte jedno pomocné

pole, do ktorého sa výsledná zotriedená postupnosť ukladá). Mergesort má časovú zložitosť $0(n \log n)$ v najhoršom aj priemernom prípade [17].

Existujú rekurzívne aj iteračné varianty. Rekurzívny mergesort je klasickým algoritmom „rozdeľuj a panuj“, ktorý rekurzívne zotriedi ľavú a pravú polovicu poľa a následne ich zlúči. Iteračný mergesort používa rovnaký proces zlučovania, ale zlučuje čiastkové úseky v inom poradí. Najskôr prechádza poľom triedených prvkov a zlučuje čiastkové úseky dĺžky jedna do čiastkových úsekov dĺžky dva. Potom zlučuje čiastkové úseky dĺžky dva do čiastkových úsekov dĺžky štyri. Následné priechody poľom vytvárajú stále dlhšie a dlhšie zotriedené čiastkové úseky až kým nevznikne jedna zotriedená postupnosť obsahujúca všetky triedené prvky.

Za základný algoritmus bol vybraný iteračný mergesort popísaný Knuthom [7], pretože je ľahko implementovateľný a prístupný tradičným optimalizačným technikám. Tento základný mergesort pracuje so vstupným poľom prvkov a používa pomocné pole na ukladanie čiastočne zlúčených úsekov. Je síce pravda, že existujú alogoritmy na zlúčenie dvoch zotriedených postupností na mieste (tzv. „in-place“) a v lineárnom čase [18], ale tieto algoritmy sú extrémne zložité a v praxi bežia veľmi pomaly. V prvej iterácií prejde základný mergesort vstupným poľom a zapisuje zlúčené úseky dĺžky dva do pomocného poľa. Úseky dĺžky dva v pomocnom poli sa následne zlúčia do úsekov dĺžky štyri a zapíšu sa do vstupného poľa. Keď si budeme pamätať, ktoré pole je zdrojové a ktoré cieľové (tzn. odkiaľ zlučujeme kam), môžeme zlučovať úseky tam a späť medzi vstupným a pomocným poľom bez potreby kopírovania. Na záver sa vykoná kontrola a ak výsledná zotriedená postupnosť skončila v pomocnom poli, skopírujeme ju naspäť do vstupného poľa.



Obr. 4.1: Usporiadanie párov čiastkových úsekov v základnom mergesorte.

Neefektívnosť iteračného mergesortu, tak ako ho popísal Knuth, je nutnosť vykonávania kontroly koncových podmienok počas každého kroku zlučovacieho procesu. Na začiatku zlučovania algoritmus pozná počet prvkov v oboch zotriedených úsekoch a teda aj počet prvkov vo výslednej postupnosti. Algoritmus ale už nevie, ako rýchlo sa spracuje každý zo zlučovaných úsekov a ktorý z úsekov sa vyčerpá skôr. V krajnom prípade sa všetky prvky z jedného úseku spotrebujú skôr ako sa vôbec použije akýkoľvek prvok z druhého úseku. Z tohto dôvodu musia jednotlivé zlučovacie kroky algoritmu kontrolovať, či neostali prvky v zlučovaných úsekoch. Je však možné zlúčiť dva úseky bez nutnosti kontroly, či boli vyčerpané všetky prvky z oboch úsekov, a to nasledujúcim spôsobom: obrátime poradie prvkov druhého úseku (teda v druhom zo zlučovaných úsekov budú prvky v opačnom poradí) a budeme zlučovať zvonku do stredu, viď.

obrázok Obr. 4.1. Keď sa jeden zo zlučovaných úsekov úplne vyčerpá, zväčšíme index úseku (v poli) a tým pádom bude ukazovať na najväčší prvok z druhého úseku. Keďže tento prvok je väčší ako každý zostávajúci prvok druhého zoznamu, slúži ako mantinel a nebude vybraný až do posledného kroku zlučovania. Táto optimalizácia vyžaduje zmenu v zlučovacom procese algoritmu tak, aby bol schopný vytvárať aj spätne zotriedené postupnosti. Jedna iterácia teda zlučuje páry úsekov a striedavo vytvára normálne a spätne zotriedené postupnosti. Touto optimalizáciou síce skomplikujeme algoritmus, ale zároveň znížime počet vykonávaných inštrukcií.

Známou optimalizáciou triediacich algoritmov je triediť dostatočne malé podpolia jednoduchým algoritmom so zložitosťou $0(n^2)$. Keď je veľkosť triedeného problému malá, jednoduché triediace algoritmy, ako napr. insertion sort, dokážu triediť rýchlejšie ako algoritmy so zložitosťou $0(n \log n)$ z dôvodu menšej konštanty. Pre základný mergesort sa autori rozhodli triediť podproblémy dĺžky štyri rýchlou inline triediacou metódou. LaMarca a Ladner na záver ešte doporučujú aplikovať metódu „rozbalovania cyklov“ (loop unrolling) na najvnútornejší cyklus algoritmu [19].

Základným mergesortom teda rozumieme Knuthov mergesort s aplikovanými optimalizáciami popísanými v predchádzajúcich odstavcoch. Tento základný mergesort vykoná $\lceil \log_2(n/4) \rceil$ priechodov vstupným poľom. Ak je počet priechodov vstupným poľom nepárny, je nakoniec treba prekopírovať zotriedené pole naspäť do vstupného poľa. Základný mergesort vykonáva menej ako polovicu inštrukcií oproti počtu inštrukcií bežného heapsortu (pozn.: bežným heapsortom sa na tomto mieste myslí Williamsov heapsort s Floydovou metódou výstavby haldy) [2].

Základný mergesort síce vykonáva malý počet inštrukcií, ale z teoretického hľadiska pamätí a cache je neefektívny. Základný mergesort použije každý prvok iba jeden krát za priechod. Ak vstupné pole presahuje veľkosť cache pamäte, prvky sa z cache vyhodia ešte skôr, ako by sa mohli znovu použiť. Ak je množina prvkov menšia alebo rovná $BC/2$, celé triedenie môže prebehnúť v cache pamäti a z cache výpadkov nastanú iba compulsory výpadky. Ak je množina prvkov väčšia ako $BC/2$, opätovná využiteľnosť prudko klesá a ak je množina kľúčov väčšia ako cache pamäť, opätovné využitie je nulové. Aby sa vylepšila popísaná neefektivita, aplikujú sa na základný mergesort dve pamäťové optimalizácie. Aplikáciou prvej pamäťovej optimalizácie vzniká algoritmus, ktorý sa nazýva *tiled mergesort.* Aplikáciou oboch pamäťových optimalizácií vzniká algoritmus, ktorý sa nazýva *multimergesort*.

Neefektívnosť základného mergesortu popísaná v predchádzajúcich odstavcoch sa môže vylepšiť tak, že priechody poľom budú dostatočne malé na to, aby sa mohli prvky znovu použiť. To je možné dosiahnuť tak, že najskôr rozdelíme vstupné pole na podmnožiny veľkosti polovice cache pamäte a potom ich zotriedime. Následne môžeme tieto zotriedené podpolia zlúčiť obyčajným zlučovacím priechodom. Táto technika sa nazýva tiling a používa sa tiež pri optimalizácií kompilátorov [9]. Tiled mergesort má teda dve fázy. Aby sa zlepšila temporal locality algoritmu, v prvej fáze sa triedia podpostupnosti dĺžky $BC/2$ použitím neoptimalizovaného obyčajného mergesortu. V druhej fáze sa vrátime k základnému mergesortu a dokončíme triedenie celého poľa. Aby sme sa vyhli záverečnému kopírovaniu ak je $\lceil \log_2(n/4) \rceil$ nepárne číslo, triedime in-line podpostupnosti dĺžky dva namiesto štyri. Aplikácia techniky tiling má za úlohu redukovať počet spôsobených cache výpadkov.

Tiling zlepšuje prvú fázu algoritmu z hľadiska cache pamätí. Druhá fáza však stále trpí tým

istým problémom ako základný mergesort. Každý priechod vstupným poľom v druhej fáze musí nutne zlyhať vo všetkých blokoch a nedosiahneme žiadnej znovupoužiteľnosti ak je množina prvkov väčšia ako cache. Aby sa napravila neúčinnosť druhej fázy, použije sa viaccestné zlievanie podobné tomu používanému pri vonkajšom triedení [7]. V multimergesorte vymeníme záverečných $\lceil \log_2(n/(BC/2)) \rceil$ zlučovacích priechodov tiled mergesortu za jediný priechod, ktorý naraz zlúči všetky časti. Tento záverečný priechod využíva pamäťovo optimalizovanú haldu, v ktorej si ukladá hlavičky zlučovaných zoznamov. Aby sme sa vyhli výpadkom cache v prípade, keď sa hlavičky zoznamov namapujú na rovnaký cache blok, načítame celý blok kľúčov (teda B kľúčov) do haldy z každého zoznamu, ktorý potrebuje ďalší prvok v halde. Halda má teda maximálnu veľkosť kb, kde k je počet zlievaných zoznamov. Aby sme sa vyhli zbytočnému kopírovaniu, je potrebné zabezpečiť, aby sa výsledok v prvej fáze algoritmu nachádzal v pomocnom poli. To sa dosiahne tak, že sa budú triediť in-line podpolia veľkosti 2 a nie 4, ak je číslo $\lceil \log_2(BC/8) \rceil$ nepárne. Viaccestné zlievanie zavádza do algoritmu určitú komplikáciu a podstatne zvyšuje počet vykonávaných inštrukcií. Na druhej strane má výsledný algoritmus výbornú cache zložitosť.

Uvedieme teoretickú aproximáciu počtu cache výpadkov pre základný mergesort a jeho varianty. Základný mergesort pre $n \leq BC/2$ spôsobí $2/B$ výpadkov na kľúč (počet compulsory výpadkov). Pretože tiled mergesort aj multimergesort používajú pre $n \leq BC/2$ základný mergesort, spôsobujú v tomto rozpätí rovnaký počet výpadkov na kľúč. Pre $n > BC/2$ je počet výpadkov na kľúč spôsobených základným mergesortom aproximovaný na

$$(2/B)\lceil \log_2(n/4) \rceil + (1/B) + (2/B)(\lceil \log_2(n/4) \rceil \bmod 2).$$

Prvá časť predchádzajúceho vzorca, $(2/B)\lceil \log_2(n/4) \rceil$, je počet capacity výpadkov spôsobených počas $\lceil \log_2(n/4) \rceil$ zlučovacích priechodov. Počas každého priechodu sa každý kľúč presunie zo zdrojového poľa do cieľového poľa. Každý B-ty kľúč, na ktorý sa pristúpi v zdrojovom poli, spôsobí jeden cache výpadok a každý B-ty kľúč zapísaný do cieľového poľa spôsobí jeden cache výpadok. Druhá časť predchádzajúceho vzorca, $1/B$, je počet compulsory výpadkov na kľúč v prvotnej fáze triedenia (in-line triedenie podpolí veľkosti 4). Posledná časť vzorca je počet výpadkov na kľúč spôsobených záverečným kopírovaním ak $\lceil \log_2(n/4) \rceil$ je nepárne.

Pre $n > BC/2$ spôsobí tiled mergesort približne $(2/B)\lceil \log_2(2n/BC) \rceil + (2/B)$ výpadkov na kľúč. Prvá časť predchádzajúceho vzorca je počet výpadkov na kľúč pre záverečných $\lceil \log_2 n/(BC/2) \rceil$ priechodov. Druhá časť vzorca je počet výpadkov na kľúč pre triedenie do častí veľkosti $BC/2$. Počet priechodov vstupným poľom je vždy párny a preto nenastávajú ďalšie cache výpadky spôsobené záverečným kopírovaním.

Pre $n > BC/2$ spôsobí multimergesort približne $4/B$ výpadkov na kľúč. Prvá fáza multimergesortu je tiled mergesort, ktorý spôsobí $2/B$ výpadkov na kľúč. V algoritme je zaručené, že počet priechodov v prvej fáze je nepárny, aby v druhej fáze prebiehalo viaccestné zlučovanie z pomocného poľa do vstupného poľa. Ak budeme ignorovať výpadky spôsobené prístupmi do haldy, druhá fáza spôsobí $2/B$ výpadkov. Multimergesort používa haldu, ktorá neobsahuje viac ako $m = B\lceil 2n/BC \rceil$ kľúčov. Z praktických dôvodov je m veľmi malé v porovnaní s veľkosťou cache pamäte. Z tohto dôvodu môžeme výpadky spôsobené prístupmi do haldy ignorovať.

4.1.2 Quicksort, memory-tuned quicksort, multiquicksort

Quicksort je populárny triediaci algoritmus založený na technike „rozdeľuj a panuj“ a princípe

rekurzie. Celý algoritmus je založený na jednoduchej myšlienke. Z množiny triedených prvkov sa vyberie jeden prvok ako pivot a ostatné prvky v poli sa preusporiadajú okolo tohto pivota tak, aby v ľavej časti boli prvky menšie ako pivot a v pravej časti prvky väčšie ako pivot. Po tomto rozdelení určite platí, že prvky ležiace v ľavej (pravej) časti poľa ostanú v ľavej (pravej) časti aj po úplnom zotriedení celého poľa. Stačí teda zotriediť samostatne ľavý a pravý úsek poľa, čo sú vlastne dve úlohy rovnakého typu ako pôvodná úloha. Následne sa teda rekurzívne postupuje pre ľavú a pravú časť pokým nezískame úseky dĺžky jedna. Tie sú samozrejme samy o sebe už zotriedené a nemusíme s nimi už nič robiť [17]. Časová zložitosť quicksortu je v priemernom prípade $0(n \log n)$, v najhoršom prípade $0(n^2)$.

Za základný quicksort bola zvolená implementáciu quicksortu poďla Sedgewicka [3]. Sedgewick doporučuje tri optimalizácie quicksort algoritmu:

- Použitie stacku namiesto rekurzie. Sedgewick radí používať iteračnú variantu quicksort namiesto rekurzívnej varianty. Pridaním pomocného stacku, v ktorom si budeme udržovať stav práce algoritmu, môžeme zmeniť rekurziu na cyklus. Táto zmena z rekurzívneho algoritmu na iteračný nemení poradie, v ktorom sa pristupuje na prvky.
- Použitie mediánu troch na výber pivota. Sedgewickove druhé doporučenie je vyberať pivota ako medián troch náhodných prvkov. Kvalita rozdeľovacieho priechodu závisí na tom, v akom pomere sa prvky množiny rozdelia do častí. Zlá voľba pivota môže spôsobiť také rozdelenie množiny, že 99% prvkov bude v jednej časti a 1% prvkov bude v druhej časti. Čím bližšie bude hodnota pivota k mediánu spracovávanej množiny prvkov, tým rovnomernejšie rozdelené budú výsledné časti a tým rýchlejšie pobeží algoritmus. Namiesto náhodného výberu pivota Sedgewick radí vyberať pivota ako medián prvého, prostredného a posledného prvku.
- Nepoužívať quicksort na triedenie malých podpostupností. Dostatočne malé podproblémy sa nechajú nezotriedené. Nakoniec sa počas záverečného priechodu zotriedi pole pomocou optimalizovaného insertion sortu.

V praxi sa quicksort vo všeobecnosti správa dobre aj z hľadiska cache zložitosti. Pretože algoritmus vykonáva sekvenčné priechody vstupným poľom, všetky kľúče v bloku sa vždy použijú. Ak je podmnožina triedeného poľa dosť malá na to, aby sa zmestila do cache, quicksort spôsobí nanajvýš jeden cache výpadok na blok skôr, ako sa celá podmnožina utriedi. Aj napriek tomuto faktu sa aplikáciou optimalizácií na základný quicksort môže docieliť lepších verzií. Autori aplikáciou optimalizácií vyvinuli dva algoritmy [2]: *memory-tuned quicksort* a *multiquicksort*

Memory-tuned quicksort jednoducho odstráni Sedgewickov insertion sort na konci. Namiesto toho triedi malé podpolia hneď, ako sa na ne narazí použitím obyčajného insertion sortu. Motivácia tohto kroku je, že keď sa počas behu algoritmu narazí na dostatočne malý podproblém, tento podproblém bol práve súčasťou prerovnávania prvkov a teda všetky prvky podproblému by sa mali nachádzať v cache pamäti. Odkladanie malých podproblémov na koniec síce dáva zmysel z hľadiska počtu vykonávaných inštrukcií, je to ale nesprávny prístup z hľadiska optimalizácie pre cache pamäte.

Multiquicksort vzniká aplikáciou podobnej optimalizácie ako u multimergesortu. Pre malé problémy, ktoré sa zmestia do cache, spôsobí quicksort iba jeden výpadok na prvok. Pre veľké problémy je počet výpadkov ale podstatne vyšší. Za účelom zlepšenia počtu výpadkov sa môže použiť multipartitioning, ktorým sa rozdelí vstupné pole na začiatku na množstvo menších

podproblémov (ktoré majú väčšiu pravdepodobnosť zmestiť sa do cache).

Multipartitioning sa používa v paralelných triediacich algoritmoch. Autori zvolili počet pivotov tak, aby počet podpolí väčších ako cache bol malý s veľkou pravdepodobnosťou. Je známe, že ak rozdelíme k bodov náhodne na úsečke o veľkosti 1, pravdepodobnosť že výsledný podúsek bude mať dĺžku x je $(1+x)^k$. Multiquicksort delí vstupné pole na $3n/BC$ kúskov, na čo je potrebných $3n/BC-1$ pivotov. Pravdepodobnosť, že podpole bude väčšie ako BC, je po rozdelení $(1-BC/n)^{(3n/BC-1)}$.

Multiquicksort vyžaduje niekoľko pomocných dátových štruktúr, nakoľko viaccestné rozdeľovanie nemôže byť efektívne vykonané in-place. Pre každý z k podproblémov sa naalokuje pomocný spojový zoznam blokov kľúčov. Počet kľúčov na blok je 100. Náhodne zvolení pivoti, ktorých je k, sa uložia do poľa. Pri načítaní prvku zo vstupného poľa sa binárnym hľadaním zistí, do ktorého zoznamu prvok patrí. V momente, keď sú všetky prvky vstupného poľa rozdelené, skopíruje sa každý zoznam naspäť do vstupného poľa a triedenie sa dokončí základným quicksort algoritmom.

Uvedieme teoretickú aproximáciu počtu cache výpadkov pre základný quicksort, memory tuned quicksort a multiquicksort. Memory tuned quicksort pre $n \leq BC$ spôsobí $1/B$ výpadkov na kľúč. Pre $n > BC$ je očakávaný počet výpadkov na kľúč aproximovaný na $2/B \ln(n/BC) + 5/(8B) + (3C)/8n$. Základný quicksort spôsobuje taký istý počet cache výpadkov na kľúč ako memory tuned quicksort plus $1/B$ cache výpadkov na kľúč.
Pre $n \leq BC$ spôsobí multiquicksort $1/B$ výpadkov na kľúč. Pre $n > BC$ je očakávaný počet výpadkov na kľúč aproximovaný na $4/B$. Podrobnejšie je aproximácia počtu výpadkov cache rozobraná v [2].

4.1.3 Heapsort, d-árny heapsort, memory tuned heapsort

S myšlienkou využiť pre úlohu triedenia dátovú štruktúru haldy prišiel prvýkrát v roku 1964 J. W. Williams, výsledný algoritmus sa nazýva heapsort. Halda je datová štruktúra, ktorá poskytuje nájdenie minimálneho prvku v konštantnom čase, odobranie minima a pridanie nového prvku v čase $0(\log n)$. Halda je binárny strom, v ktorom sú splnené nasledujúce podmienky:

1. V každej hladine od prvej až do predposlednej je maximálny možný počet uzlov (v k-tej hladine je 2^{k-1} uzlov).
2. V poslednej hladine sú všetky uzly umiestnené čo možno najviac vľavo.
3. Pre každý uzol platí, že hodnota v ňom uložená je menšia ako hodnota uložená v jeho ľubovoľnom následníkovi.

Algoritmus heapsort triedi dáta postavením haldy (začneme s primitívnou haldou obsahujúcou iba prvé z triedených čísiel a postupne do tejto hlady vložíme zostávajúce triedené čísla), obsahujúcej všetky prvky vstupného poľa a následným odoberaním minima z koreňa stromu. Halda sa dá reprezentovať v poli, čím sa z heapsortu stáva jednoduchý triediaci algoritmus. V roku 1964 navrhol R. W. Floyd vylepšenú techniku výstavby haldy zdola od listov, ktorá má lineárnu časovú zložitosť [20]. Zložitosť heapsortu je v najhoršom aj priemernom prípade $0(N \log N)$.

Za základný heapsort bol zvolený Williamsov algoritmus s Floydovou metódou výstavby haldy. Pretože v základnom heapsorte môže mať rodič dvoch, jedného alebo žiadnych synov, základný heapsort potrebuje vykonať dve kontroly pre každú úroveň haldy: musí skontrolovať, či prvok má

vôbec nejakých synov, a ďalšiu kontrolu, či má iba jedného syna. Toto sa dá zredukovať na jednu kontrolu, ak zaručíme, že každý rodič bude mať buď dvoch alebo žiadneho syna. To dosiahneme tak, že maximálny prvok vždy uložíme za posledný prvok haldy v poli. Týmto spôsobom to bude vypadať akokeby rodičia s jedným synom mali vlastne dvoch: jedného svojho syna a maximálny prvok, ktorý ale nikdy nebude vybraný za minimum. V dôsledku tejto zmeny bude možno treba vykonať v poslednej vrstve haldy jedno porovnanie naviac, ale všetky ostatné vrstvy budú potrebovať už iba jednu kontrolu na to, či majú synov.

Výsledky [10] ukazujú, že niektoré optimalizácie háld zlepšujú ich cache zložitosť. Tieto optimalizácie budú aplikované na základný heapsort za účelom zlepšenia jeho výkonu.
Pamäťové optimalizácie pre heapsort zahŕňajú nahradenie binárnej haldy *B*-haldou, kde *B* je počet kľúčov, ktoré sa zmestia do jedného cache bloku. Všeobecne, v *d*-halde [8] má každý uzol, okrem listov, namiesto dvoch synov *d* synov. Ak je *B* relatívne malé, napr. 4 alebo 8, znižuje sa aj počet vykonávaných inštrukcií pre operácie pridávania a odoberania maxima. Ďalšou optimalizáciou je zarovnať haldu v pamäti tak, aby všetci synovia (ktorých počet je B) boli v jednom cache bloku. Algoritmus využívajúci prvú z optimalizácií nazveme *d-árny heapsort* a algoritmus využívajúci obe optimalizácie nazveme *memory tuned heapsort*.

Teoretická aproximácia počtu cache výpadkov pre heapsort a jeho varianty je netriviálna a neexistuje pre ňu jednoduchý vzorec ako to bolo v prípade napr. mergesortu. V prípade záujmu sa môže čitateľ obrátiť na [2] a na [10].

4.2 Funnelsort

Cache-oblivious algoritmy, ako napr. mergesort, nie sú optimálne vzhľadom k počtu cache výpadkov. Táto sekcia popisuje triediaci cache-oblivious algoritmus, nazývajúci sa funnelsort, ktorý bol popísaný Frigom a spol. [4]. Frigo a spol. skúmali funnelsort v ideal-cache modeli. Funnelsort má optimálnu časovú zložitosť $0\left(n \log n\right)$ a optimálnu cache zložitosť $O(1+\left(n/L\right)(1+\log_z n))$.

Funnelsort je podobný mergesortu. Na zotriedenie súvislého poľa n prvkov vykonáva nasledujúce dva kroky:

1. Rozdeľ vstupné pole na $n^{1/3}$ súvislých polí veľkosti $n^{2/3}$ a zotrieď tieto polia rekurzívne.
2. Zlej $n^{1/3}$ zotriedených postupností použitím $n^{1/3}$-mergeru, ktorý je popísaný nižšie.

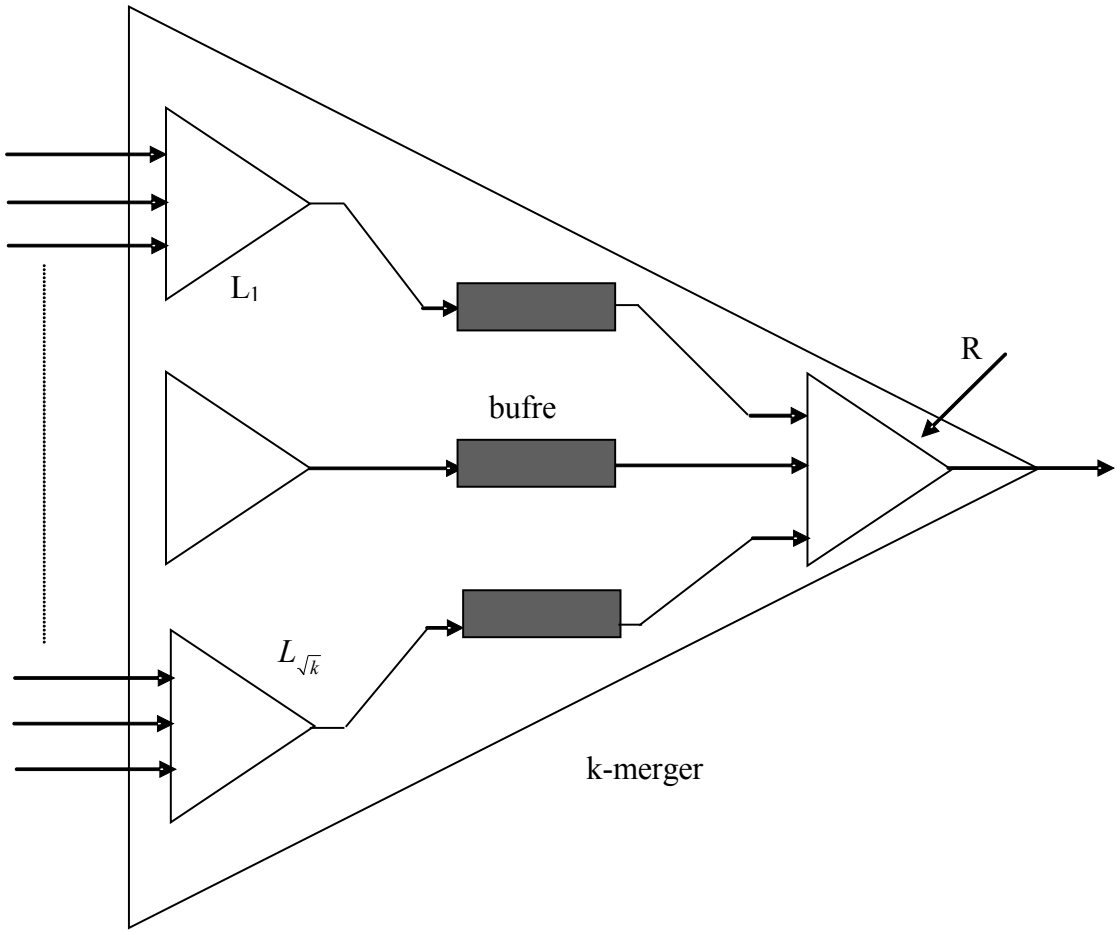
Funnelsort sa líši od mergesortu v spôsobe, akým funguje (akým sa vykonáva) zlievanie. Zlievanie sa vykonáva zariadením nazývaným *k*-merger, ktoré na vstupe dostane k zotriedených postupností a zleje ich. *K*-merger pracuje tak, že rekurzívne zlieva zotriedené postupnosti, ktoré sa postupne stávajú dlhšími. Na rozdiel od mergesortu, *k*-merger pozastaví prácu na zlievaní podproblému, keď výstupná postupnosť dosiahne „dostatočnú“ veľkosť a začne pracovať na zlievaní iného podproblému. Na obrázku Obr. 4.2 je zobrazenie *k*-mergeru.

Počas práce *k*-merger udržuje nasledujúci invariant: Každé zavolanie *k*-mergeru vyprodukuje nasledujúcich k^3 prvkov zotriedeného poľa, získaného zlievaním k vstupných postupností.

K-merger je vybudovaný rekurzívne z $\sqrt{k}$-mergerov nasledujúcim spôsobom: k vstupov je rozdelených do $\sqrt{k}$ množín po $\sqrt{k}$ prvkoch, ktoré tvoria vstup pre $\sqrt{k}$ $\sqrt{k}$-mergerov $L_1, L_2, ..., L_{\sqrt{k}}$ v ľavej časti obrázku Obr. 4.2. Výstupy týchto mergerov sú napojené na vstupy $\sqrt{k}$

bufferov. Každý buffer je FIFO veľkosti $2k^{3/2}$. Výstupy bufferov sú nakoniec napojené na $\sqrt{k}$ vstupov $\sqrt{k}$ -mergeru R v pravej časti obrázku Obr. 4.2. Výstup tohto posledného $\sqrt{k}$ -mergeru je výstupom celého *k*-mergeru. Buffery sú predimenzované, každý môže poňať $2k^{3/2}$ prvkov, čo je dvojnásobok $k^{3/2}$ prvkov vyprodukovaných jedným $\sqrt{k}$ -mergerom. Toto predimedzovanie je potrebné pre správny chod algoritmu ako bude ukázané ďalej. Konečným prípadom rekurzie je *k*-merger s k rovným 2, ktorý vyprodukuje $k^3 = 8$ prvkov.

K-merger pracuje rekurzívne. Aby vyprodukoval k^3 prvkov, *k*-merger zavolá R $k^{3/2}$ krát. Pred každým zavolaním *k*-merger naplní všetky buffre, ktoré sú menej ako spolovice plné (tzn. všetky buffre, ktoré obsahujú menej ako $k^{3/2}$ prvkov). Aby sme mohli naplniť buffer i, algoritmus zavolá odpovedajúci „ľavý“ merger L_i jeden krát. Keďže L_i vyprodukuje $k^{3/2}$ prvkov, buffer obsahuje aspoň $k^{3/2}$ prvkov po tom, ako L_i skončí.



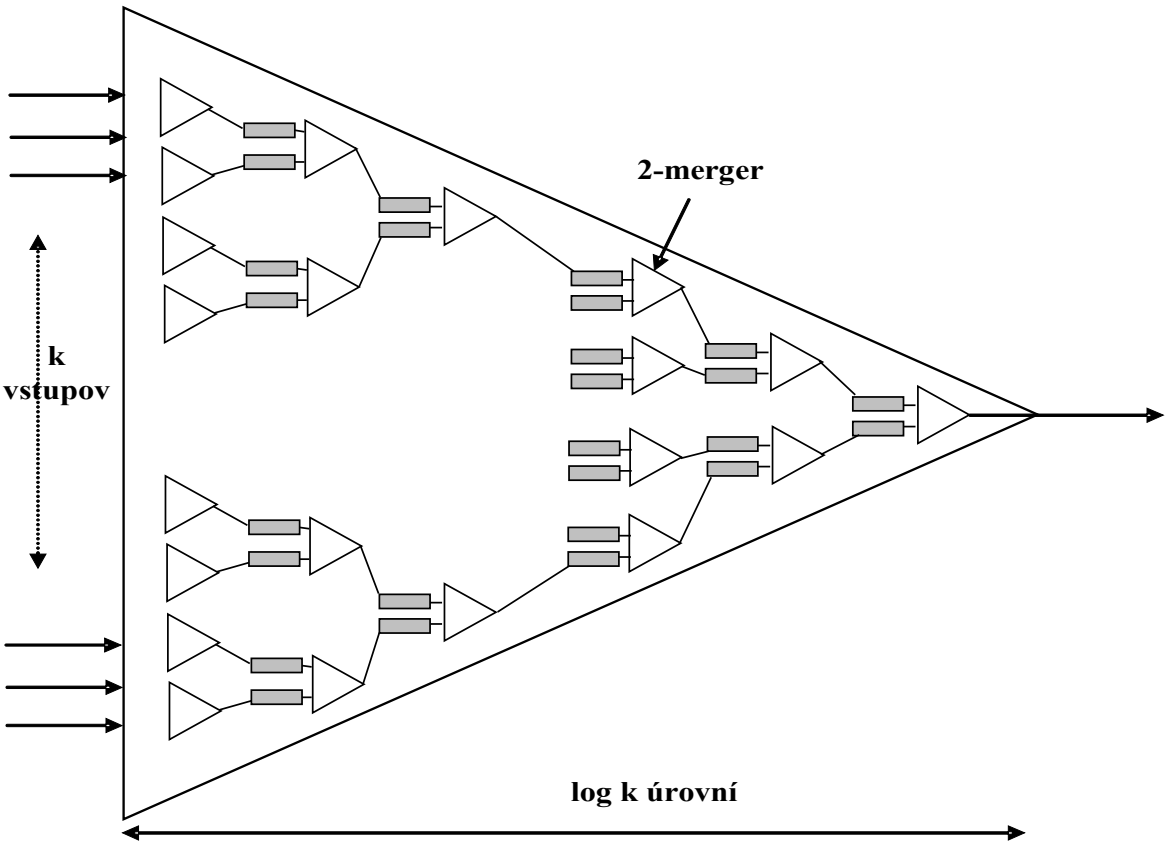
Obr. 4.2: *K*-merger je vystavaný rekurzívne z $\sqrt{k}$ „ľavých“ $\sqrt{k}$ - *mergerov* $L_1, ..., L_{\sqrt{k}}$, zo skupiny bufferov a jedného „pravého“ $\sqrt{k}$ - *mergeru* R.

Dá sa indukciou dokázať, že zložitosť funnelsortu je $0(n \log n)$. Pre cache zložitosť funnelsortu uvedieme náznak dôkazu. Pre $n < \alpha Z$, kde α je dostatočne malá konštanta, sa algoritmus zmestí do cache pamäte (najväčší *k*-merger je $n^{1/3}$-merger, ktorý potrebuje $O(n^{2/3}) < O(n)$ priestoru, a v každom momente je aktívny práve jeden *k*-merger). Algoritmus v tomto prípade spôsobí $O(1 + n/L)$ cache výpadkov. Ak je $n > \alpha Z$, dostávame rekurziu $Q(n) = n^{1/3} Q(n^{2/3}) + Q_M(n^{1/3})$. Na tomto mieste v dôkaze sa používa pomocné tvrdenie: *K*-merger pri práci spôsobí nanajvýš $Q_M(k)$ výpadkov cache pamäte, za predpokladu, že cache pamäť je „vysoká“ (t.j. $Z = \Omega(L^2)$), kde $Q_M(k) = O(1 + k + k^3/L + k^3 \log_Z k/L)$.

Z pomocného tvrdenia dostávame $Q_M(n^{1/3}) = O(1 + n^{1/3} + n/L + n \log_z n/L)$. Z predpokladu tvrdenia, teda že $Z = \Omega(L^2)$, dostávame $n/L = \Omega(n^{1/3})$. Ďalej platí, že $n^{1/3} = \Omega(1)$ a $\log n = \Omega(\log Z)$. Z toho vyplýva, že $Q_M(n^{1/3}) = O(n \log_z n/L)$ a teda rekurzia sa môže zjednodušiť na $Q(n) = n^{1/3} Q(n^{2/3}) + O(n \log_z n/L)$. Následne sa postupuje indukciou podľa *n*. Pre podrobnosti dôkazu viď. [4].

4.2.1 Lazy unnelsort

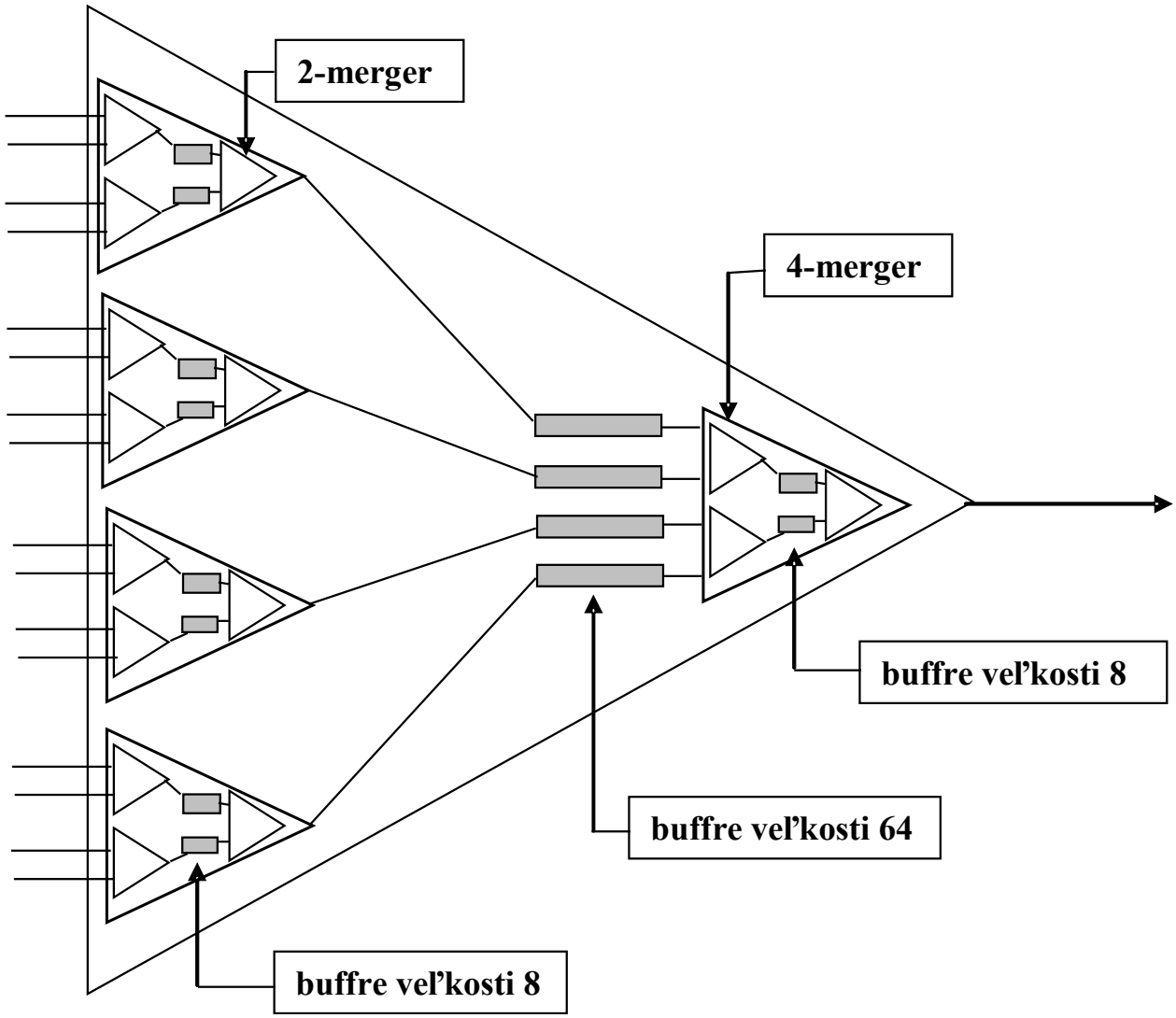
Lazy funnelsort je varianta triediaceho algoritmu funnelsort. Autormi sú G.S.Brodal a R.Fagerberg [11]. Výhoda lazy funnelsortu je, že jeho popis, analýza a implementácia sú oproti pôvodnému algoritmu jednoduchšie. Pre popis algoritmu zaviedli autori parameter d . Tento parameter pre $d = 3$ súhlasí s popisom pôvodného funnelsortu.



Obr. 4.3: *K*-merger rozvinutý do 2-mergerov.

Hlavným prvkom funnelsortu je koncept *k*-mergeru, ktorý na každé zavolanie zlúči nasledujúcich k^d prvkov z *k* zotriedených vstupných postupností. Pretože *k*-merger potrebuje pamäť, ktorej veľkosť rastie exponenciálne vzhľadom ku *k*, nie je možné zlievať všetkých *n* prvkov vstupného poľa *n*-mergerom. Funnelsort rekurzívne vyprodukuje $n^{1/d}$ zotriedených postupností veľkosti $n^{1-1/d}$ a následne ich zleje použitím $n^{1/d}$-mergeru. Zavolanie k-mergeru v pôvodnom funnelsorte zahrňuje rozvrhovanie sub-mergerov podľa kontroly zaplnenia všetkých jeho bufferov v daných intervaloch.

Hlavná zmena lazy funnelsortu oproti funnelsortu spočíva v zoslabení požiadavky, aby boli všetky buffre kontrolované a v prípade potreby naplnené v rovnakom čase. V lazy funnelsorte sa buffer naplní až po tom, ako sa vyprázdni. Táto zmena umožnuje rozvinúť rekurzívnu definíciu *k*-mergeru do stromu binárnych mergerov, v ktorom hrany predstavujú buffre a definovať v algoritme prácu mergeru priamo pomocou uzlov tohto stromu. Pre potreby tohto algoritmu sa definuje *k*-merger ako úplný vyvážený binárny strom s *k* listami. Postupnosti prvkov budeme hovoriť tiež „stream“. Každý list obsahuje, resp. v liste je pripojený, zotriedený vstupný stream a každý vnútorný uzol obsahuje binárny merger. Výstup koreňa stromu je výstupný stream celého *k*-mergeru.



Obr.4.4: 16-merger rozvinutý do 2-mergerov s veľkosťami bufferov pre jednotlivé úrovne.

Každá hrana medzi dvoma vnútornými uzlami obsahuje buffer, ktorý plní úlohu výstupného streamu pre merger v uzle na nižšej úrovni a je zároveň jedným zo vstupných streamov pre merger v uzle na vyššej úrovni.

Na obrázku Obr. 4.3. je zobrazený *k*-merger rozvinutý do 2-mergerov. Veľkosti bufferov sa definujú rekurzívne. Nech $D_0 = \lceil \log(k)/2 \rceil$ označuje číslo prostrednej úrovne stromu, označme ako vrchný strom podstrom pozostávajúci zo všetkých uzlov úrovne nanajvýš D_0 a označme všetky podstromy zakorenené v uzloch úrovne $D_0 + 1$ ako spodné stromy ($\log_a b$ je na tomto mieste použitý v zmysle $\max\{1, \log_x y\}$). Hrany medzi uzlami v hĺbke D_0 a uzlami v hĺbke $D_0 + 1$ obsahujú buffre veľkosti $\lceil k^{d/2} \rceil$.

Veľkosti ostatných bufferov sú definované rekurzívne pre vrchný strom a spodné stromy. Výstupný stream koreňa stromu, a teda celého *k*-mergeru, má veľkosť k^d. Na obrázku Obr. 4.4. je zobrazený 16-merger s veľkosťami bufferov pre jednotlivé úrovne. V [11] Brodal a Fagerberg uvádzali, že *k*-merger musí byť v pamäti uložený v súlade so špecifickým rozložením, ktoré sa nazýva Van Emde Boas Layout [24]. V [28] však tí istí autori popierajú nutnosť špecifického uloženia štruktúry *k*-mergeru v pamäti. Dôkaz je možné nájsť v [28].

Na obrázku Obr. 4.5 je zobrazený algoritmus binárneho zlievania pre každý vnútorný uzol *k*-mergeru. Posledný riadok znamená presunúť menší z dvoch prvkov zo začiatku vstupných bufferov na koniec výstupného bufferu. Celý *k*-merger sa spustí zavolaním funkcie FILL() na koreň *k*-mergeru. Počas práce *k*-mergeru dôjde k vyčerpaniu vstupných bufferov. Vyčerpanie vstupných prvkov by sa malo propagovať v *k*-mergeri smerom nahor tak, že buffer označíme ako „vyčerpaný“ práve vtedy, keď sú oba jeho vstupné buffre označené ako „vyčerpané“. Toto je jednoduché rozšírenie algoritmu na Obr. 4.5. Každý buffer sa napĺňa novými prvkami až vtedy, keď sa úplne

```
funkcia FILL ( v )
     while výstupný buffer uzlu v nie je plný
          if ľavý vstupný buffer je prázdny
               FILL(ľavý syn v)
          if pravý vstupný buffer je prázdny
               FILL(pravý syn v)
          vykonaj jeden zlievací krok
```

Obr.4.5: Algoritmus binárneho zlievania pre lazy funnelsort

vyprázdni, čo odstraňuje nutnosť implementovať buffre ako kruhové fronty, ako tomu bolo u funnelsortu.

Autori algoritmu trochu nešťastne zvolili pre analýzu algoritmu I/O model, popísaný v Sekcii 3.1. Pre pripomenutie, I/O model predpokladá pamäťovú hierarchiu pozostávajúcu z dvoch úrovní. Menšia z úrovní má veľkosť M a prenosy medzi úrovňami sa vykonávajú po blokoch veľkosti B prvkov. O druhej z pamäťových úrovní neuvažujú ako o disku. Cache-oblivious algoritmus chápu ako algoritmus optimalizovaný v I/O model s tým rozdielom, že sa optimalizuje na veľkosť bloku B a veľkosť pamäte M, ktoré ale nie sú známe. Ako odôvodnenie sa uvádza, že pokiaľ analýza platí pre akúkoľvek veľkosť bloku a veľkosť pamäte, platí pre všetky úrovne pamäťovej hierarchie.

Bez dôkazu uvedieme priestorovú zložitosť a časovú zložitosť lazy funnelsortu v I/O modeli:

- Nech $d \geq 2$. Veľkosť k-mergeru, bez jeho výstupného bufferu, je ohraničená číslom $ck^{(d+1)/2}$, pre konštantu $c \geq 1$. Za predpokladu $B^{(d+1)/(d-1)} \leq M/2c$, k-merger vykoná $O\left(\left(k^d/B\right)\log_M\left(k^d\right)+k\right)$ I/O operácií.
- Za rovnakých predpokladov ako v predchádzajúcom bode, lazy funnelsort potrebuje $O\left(\left(dN/B\right)\log_M N\right)$ I/O operácií na zotriedenie N prvkov.

Dôkaz je možné nájsť v [11].

4.3 Distribution Sort

V tejto sekcií je popísaný cache-oblivious algoritmus distribution sort [4]. Tak ako funnelsort, distribution sort bol skúmaný v ideal-cache modeli, jeho časová zložitosť je $0\left(n\log n\right)$ a jeho cache zložitosť je $O(1+(n/L)(1+\log_z(n)))$.

Pre dané pole A (uložené v súvislých pamäťových miestach) dĺžky n, distribution sort pracuje následovne:

1. Rozdeľ pole A do $\sqrt{n}$ súvislých podpolí dĺžky $\sqrt{n}$. Rekurzívne zotrieď každé podpole.
2. Rozdeľ zotriedené podpolia do q priehradok $B_1,\ldots B_q$ veľkosti $n_1,..,.n_q$ tak, že platí:
   a) $\max\{x \mid x \in B_i\} \leq \min\{x \mid x \in B_{i+1}\}$ pre i = 1, 2, .., q-1
   b) $n_i \leq 2\sqrt{n}$ pre i = 1, 2, ..., q
   (viď ďalej.)
3. Rekurzívne zotrieď každú priehradku.
4. Skopíruj zotriedené priehradky do poľa A.

Autori algoritmu doporučujú použiť stack-based alokátor pamäti za účelom zlepšenia priestorovej lokality.

Cieľom kroku č.2 je rozdistribuovať zotriedené podpolia poľa A do q priehradok $B_1,\ldots B_q$. Algoritmus udržuje počas svojho behu dva invarianty.

- Prvý invariant: V ktoromkoľvek momente každá priehradka obsahuje najviac $2\sqrt{n}$ prvkov a ktorýkoľvek prvok v priehradke B_i je menší ako ktorýkoľvek prvok v priehradke B_{i+1}.
- Druhý invariant: Každá priehradka má priradeného pivota. Spočiatku existuje iba jedna prázdna priehradka s pivotom rovným ∞.

Myšlienka algoritmu je skopírovať všetky prvky z podpolí do priehradok za stáleho udržovania invariantov. Pre každé podpole a priehradku sa ukladajú stavové informácie. Stavová informácia podpoľa pozostáva z indexu *next* nasledujúceho prvku, ktorý sa bude čítať z podpoľa a z čísla priehradky *bnum*, kam sa má tento prvok kopírovať. Nech číslo *bnum* je nekonečno práve vtedy keď všetky prvky v podpoli boli skopírované. Stavová informácia priehradky pozostáva z pivota a z čísla udávajúceho aktuálny počet prvkov v priehradke (teda z počítadla prvkov).

Chceli by sme skopírovať prvok na pozícií *next* z podpoľa do priehradky bnum. Ak je tento prvok väčší ako pivot priehradky *bnum*, budeme inkrementovať *bnum* dovtedy kým nenájdeme priehradku, pre ktorú je prvok menší ako jej pivot. Nanešťastie táto základná stratégia je nevhodná

z hľadiska práce s cache pamäťou a je treba použiť komplikovanejší postup.

Distribučný krok (krok č.2) dosiahneme rekurzívnou procedúrou `DISTRIBUTE(i,j,m)`, ktorá rozdeľuje prvky z *i*-tého až (*i* + *m* -1)-tého podpoľa do priehradok začínajúc od B_j. Vzhľadom k predpokladu, že každé podpole $i, i+1, ..., i+m-1$ má hodnotu $bnum \geq j$, vykonanie procedúry `DISTRIBUTE(i,j,m)` zaručí, že podpolia $i, i+1, ..., i+m-1$ budú mať $bnum \geq j+m$. Kroč č.2 distribution sortu zavolá `DISTRIBUTE(1,1,`$\sqrt{n}$`)`. Následuje pseudokód rekurzívnej implementácie procedúry `DISTRIBUTE`:

```
DISTRIBUTE (i, j, m)
IF m = 1
      THEN COPYELEMS (i, j)
      ELSE  DISTRIBUTE (i, j, m/2)
            DISTRIBUTE (i + m/2, j, m/2)
            DISTRIBUTE (i, j + m/2, m/2)
            DISTRIBUTE (i + m/2, j + m/2, m/2)
```

Procedúra `COPYELEMS(i,j)` skopíruje všetky prvky z podpoľa i, ktoré patria priehradke j. Ak má po vložení priehradka j viac ako $2\sqrt{n}$ prvkov, rozdelíme ju na dve priehradky veľkosti aspoň $\sqrt{n}$. Pre operáciu rozdelenia priehradky sa odporúča použiť deterministický algoritmus na hľadanie mediánu [12] a následné rozdelenie.

Formálny dôkaz časovej a cache zložitosti algoritmu je možné nájsť v [4], na tomto mieste uvedieme len neformálny náznak. Opiera sa o pomocné tvrdenie: Distribučný krok (krok č.2) má časovú zložitosť $0(n)$, cache zložitosť $0(1+n/L)$ a na rozdelenie n prvkov využíva $O(n)$ priestoru na stacku. Časová zložitosť algoritmu je daná rekurziou:

$$W(n) = \sqrt{n}W(\sqrt{n}) + \sum^{q} W(n_i) + O(n),$$

kde pre každé n_i platí $n_i \leq 2\sqrt{n}$ a $\sum n_i = n$. Riešením tejto rekurzie je $W(n) = O(n \log n)$. Cache zložitosť algoritmu je daná rekurziou:

$$Q(n) \leq O(1+n/L), \text{ ak } n \leq \alpha Z$$

$$Q(n) \leq \sqrt{n}Q(\sqrt{n}) + \sum^{q} Q(n_i) + O(1+n/L), \text{ ak } n > \alpha Z,$$

kde α je konštanta dostatočne malá tak, aby sa miesto, ktoré potrebuje algoritmus na zásobníku pre problém veľkosti αZ spolu so vstupným poľom, zmestilo do cache pamäte. Prvý prípad rekurzívneho zápisu $n \leq \alpha Z$ nastáva, keď sa vstupné pole A spolu s priestorom potrebným pre zásobník, zmestia do $O(1+n/L)$ blokov cache pamäte. V takom prípade algoritmus spôsobí $O(1+n/L)$ výpadkov. V prípade keď $n > \alpha Z$, rekurzívne volania v krokoch jedna a tri spôsobia

$$Q(\sqrt{n}) + \sum^{q} Q(n)$$

výpadkov. $O(1+n/L)$ je cache zložitosť krokov dva a štyri, podľa pomocného tvrdenia. Na záver stačí vyriešiť rekurzívnu rovnicu, pre podrobnosti viď [4].

Kapitola 5

Realizácia testov

Cieľom testovania počítačových programov je najčastejšie získanie empirických výsledkov, ktoré podporujú alebo vyvracajú nejakú hypotézu, napr. program A vykoná menej porovnaní ako program B, alebo program A je rýchlejší ako program B.

Bez ohľadu na to, akú hypotézu sa človek, vykonávajúci testy, snaží podporiť (alebo zamietnuť), je dôležité, aby merania boli zároveň spoľahlivé, platné a reprodukovateľné. Meranie je spoľahlivé ak je porovnateľné – nezávislé merania programu súhlasia. Spoľahlivé meranie je možné dosiahnuť opakovaním merania viacej krát a ideálne na viacerých počítačoch. Meranie je platné, ak nám hovorí niečo o tom, čo sme sa snažili meraním sledovať.

V Sekcii 5.1 pojednávame o metodike testov, meraní času, prostredí testov, množine testovacích dát vybraných triediacich algoritmoch. Cieľom testov nebolo navzájom porovnať všetky možné známe triediace algoritmy, ale v praxi naimplementovať a reálne overiť pár vybraných algoritmov. V Sekcii 5.2 sa venujeme problematike výberu priemernej hodnoty a v Sekcii 5.3 rozoberáme výsledky testov.

5.1 Metodika testov

Na prípravu a spúšťanie testov je možné nahliadať ako na výzkum nejakého aspektu sledovaných programov. Aspekty, o ktorých si myslíme, že nám niečo povedia o sledovanej hypotéze ovplyvňujú spôsob konštrukcie a vykonávania testov. Preto je dôležité, zo všetkého najskôr, položiť si cieľ výzkumu.

V tejto práci máme záujem získať empirické výsledky, ktoré nám pomôžu zodpovedať otázku: Je cache-oblivious a cache-aware prístup k triedeniu dát vo vnútornej pamäti v praxi úspešný? Túto otázku pokladáme z pohľadu obyčajného užívateľa, ktorý nemá o systéme žiadne zvláštne znalosti, iba vie, že na danom počítači je cache pamäť. Tohto užívateľa samozrejme zaujíma, aký algoritmus by mal zvoliť, ak chce čo najrýchlejšie triediť dáta vo vnútornej pamäti.

Na zodpovedanie tejto otázky sme implementovali cache-aware aj cache-oblivious algoritmy a merali sme čas behu algoritmov, počet vykonaných porovnaní a kopírovaní. Bohužiaľ nám nie je známy spôsob akým by sa dali merať cache výpadky, ktoré nastávajú v pamäťovom systéme. Bol

by to určite veľmi zaujímavý aspekt porovnávacích testov. Pretože sa na vec dívame z pohľadu obyčajnéhu užívateľa, ide nám predovšetkým o skutočný čas, za ktorý daný algoritmus dokáže zotriediť dáta. Tzn. či má zmysel (a ak áno, tak za akých podmienok) implementovať zložité cache-aware a cache-oblivious algoritmy (cache-aware algoritmy dokonca aj ladiť) alebo je lepšie použiť niektorý zo základných a známych algoritmov?

5.1.1 Meranie času

Meranie času behu programu je na počítačových systémoch všeobecne problematické. Neexistuje metóda, prenosná medzi platformami, ktorá by zaručila presné zmeranie času behu daného programu. Dôvod, prečo je tomu tak, spočíva jednak v rôznosti operačných systémov, ale hlavne v tom, že jadro operačného systému pri súčasnom behu viacerých programov musí nutne procesy preplánovávať a teda zmeraný čas behu programu spravidla zahrňuje aj časti behov iných procesov. Ideálne by bolo vytvoriť nepreemptívny proces, ktorého kritické časti merania nemôžu byť preplánované prerušením. V takom prípade by namerané hodnoty boli presné a použiteľné. Presne z tohto dôvodu bol pre testy zvolený operačný systém Linux. Vyššie popísaný spôsob merania totiž môžeme ľahko dosahnuť následovne:

- Nabootovaním cez `init=/bin/sh`, čím nám systém pobeží len s nevyhnutnými službami. Výsledkom je minimálne zaťažený systém.
- Zamaskovaním prerušení, čím dosiahneme nepreemptívnosť procesu. Teda ak by aj jadro chcelo náš meraný program preplánovať, nemôže sa tak stať počas merania kritického úseku.

Technicky sa to dá docieliť pomocou funkcie `iopl(3)`, ktorou sa prepneme do príslušného runlevelu (čo potrebujeme k zakázaniu prerušení) a následným zakázaním prerušení volaním `asm("CLI")`. Po skončení samotného výpočtu môžeme bezpečne povoliť prerušenia pomocou `asm("STI")`. Stačí nám chrániť výpočtové časti programu.

Samotné meranie času sa dá robiť viacerými spôsobmi. Pôvodne sme zamýšľali merať čas pomocou počítania cyklov procesoru, tak ako to navrhuje [5]. Táto metóda vyžaduje funkciu v nasledujúcom zmysle:

```
void access_counter(unsigned *hi, unsigned *lo)
{
 asm("rdtsc; movl %%edx,%0; movl %%eax, %1"
     : "=r" (*hi), "=r" (*lo)
     : /**/
     : "%edx", "%eax");
}
```

Teda pred meraným a po meranom úseku zaznačíme stav počítadla a odčítame. Táto metóda je veľmi presná ale nie je prenosná a stále trpí nepresnosťami v prípade veľkého zataženia systému. Preto sme sa rozhodoli pre meranie funkciou `clock_gettime()` s flagom `CLOCK_PROCESS_CPUTIME_ID`, ktorá je skoro rovnako presná (presnosť v nanosekundách) a je prenosná. Nakoľko sme vyššie popísanou metódou zaručili, že proces pobeží bez prerušenia s maximom dostupných systémových prostriedkov, je táto metóda merania času dostačujúca.

5.1.2 Testovacie prostredia

V ideálnom prípade by sa testy mali vykonať na viacerých počítačoch od rôznych výrobcov, čím sa pokryjú odlišnosti rôznych architektúr a trendy moderných procesorov a pamäťových systémov. Naše testovacie počítače sú:

Počítač č.1

Procesor Intel Celeron D336
- frekvencia 2,8 GHz
- L1 cache 16+16 kB, 4-cestná množ. asoc., 64 bytové cache riadky
- L2 cache 256 kB, 8-cestná množ. asoc., 128 bytové cache riadky

Pamäť 2,5 GB DDR2
Matičná doska ASUS P5LD2 SE
OS Gentoo Linux, verzia jadra 2.6.2.2, prekladač gcc verzia 4.1.2

Počítač č.2

Procesor Intel Core 2 Duo E7300
- frekvencia 2,66 GHz
- L1 cache 32+32kB (per core), 4-cestná množ. asoc., 64 bytové cache riadky
- L2 cache 3MB shared, 8-cestná množ. asoc., 128 bytové cache riadky

Pamäť 2,5 GB DDR2
Matičná doska GA-G31M-S2L
OS 64 bitový Gentoo Linux, verzia jadra 2.6.2.4-r5, prekladač gcc verzia 4.1.2

Počítač č.3

Procesor AMD Phenom X4 9650
- frekvencia 2,3 Ghz
- L1 cache 64+64kB (per core), 2-cestná množ. asoc., 64 bytové cache riadky
- L2 cache 512kB (per core), 8-cestná množ. asoc., 64 bytové cache riadky
- L3 cache 2MB shared, 32-cestná množ. asoc., 64 bytové cache riadky

Pamäť 2,5 GB DDR2
Matičná doska MSI K9A2 CF
OS 64 bitový Gentoo Linux, verzia jadra 2.6.2.4-r5, prekladač gcc verzia 4.1.2

Počítač č.4

Procesor AMD Athlon 64 3500+
- frekvencia 2,2 Ghz
- L1 cache 64+64kB, 2-cestná množ. asoc., 64 bytové cache riadky
- L2 cache 512kB, 16-cestná množ. asoc., 64 bytové cache riadky

Pamäť 2,5 GB DDR
Matičná doska MSI K9A2 CF
OS 64 bitový Gentoo Linux, verzia jadra 2.6.2.4-r5, prekladač gcc verzia 4.1.2

Ako je vidieť, naše testovacie počítače pokrývajú dvoch najrozšírenejších výrobcov procesorov. Testovacie počítače sú nám plne k dispozícii, takže máme plnú kontrolu nad tým aké procesy na počítačoch bežia, akým spôsobom sa spúšťajú testy a vieme zaručiť, že systém je pre úlohu vykonania testov minimálne zaťažený.

5.1.3 Testovacie dáta

Bežnou praxou pri testovaní triediacich algoritmov je použiť generátor permutácií. Vstupom generátoru je dĺžka permutácie N a nepovinne hodnota, ktorá inicializuje generátor náhodných čísel. Generátor naalokuje pole A dĺžky N a inicializuje ho hodnotami od 0 do $N-1$.

Vyššie popísaný generátor permutácií sa používa hlavne vtedy, keď je cieľom testov porovnať teoretický počet porovnaní (kopírovaní, výmen...) s prakticky nameranými hodnotami. My sme sa však rozhodli pre iný prístup a to z dôvodov:

- Nám nejde o overovanie teoretického predpokladaného počtu porovnaní, kopírovaní alebo výmen.
- Myslíme si, že náhodne generované dáta predstavujú lepšiu sadu testovacích dát ako generovanie permutácií, pretože sa viac blížia reálnym dátam, na ktoré môže algoritmus naraziť v praxi.
- Triedime okrem celých čísel aj reťazce a tie sa ako permutácie generovať nedajú.

V tejto práci sme sa rozhodli realizovať testy na dvoch typoch vstupných dát, a to integre (celé čísla) a reťazce veľkosti 128 bytov. Ako generátor náhodných čísel bola zvolená funkcia `lrand48()` ktorá generuje pseudo-náhodné čísla rovnomerne distribuované v rozmedzí $(0, 2^{31})$. Ako seed bol použitý čas. Reťazce sa generujú následovne: polovica reťazcov vstupného poľa je vygenerovaná úplne náhodne (použije sa vyššie uvedená funkcia) a druhá polovica vstupného poľa je vygenerovaná tak, aby prvých 127 bytov bolo rovnakých a reťazce sa líšili až v poslednom byte. To do určitej miery odráža reálne dáta, na ktoré môže v praxi triediaci algoritmus naraziť, ako napr. url adresy, ktoré majú nejakú začiatočnú časť rovnakú. Je to vhodné aj z toho dôvodu, aby operácia porovnania v prípade reťazcov trvala dlhšie ako operácia porovnania v prípade celých čísel. Ak by sme všetky reťazce generovali náhodne, je veľmi veľká pravdepodobnosť, že väčšina reťazcov by sa od seba líšila už v niektorých prvých bytoch.

Na každom testovanom počítači sme vykonali meranie dvanástich bodov pre celé čísla a pre reťazce. Prvé štyri body majú veľkosti 1,2,8 a 32 násobku veľkosti L2 cache pamäte. Druhé štyri body sú volené ako najbližšie väčšie číslo zaokrúhlené na milióny s nejakým rovnomerným krokom, napr. 5 alebo 10 mil. prvkov pre celé čísla (ako bude uvedené ďalej, reťazce veľkosťou vstupného poľa odpovedajú celým číslam). Posledné štyri body sú volené tak, aby bol posledný bod maximum čo sa ešte zmestí do vnútornej pamäte. Jednotlivé body testov boli volené tak, aby veľkosť vstupného poľa celých čísel odpovedala veľkosti vstupného poľa reťazcov, teda ak napr. v prvom bode triedime pole celých čísel veľkosti 1 MB, aby aj odpovedajúce pole reťazcov malo veľkosť 1 MB.

Uvedieme príklad pre veľkosti bodov na počítači č.1. Prvých osem bodov testov vypadá následovne, v prvom riadku sú celé čísla, v druhom sú reťazce:

65536	131072	524288	2097152	5 mil.	10 mil.	15 mil.	20 mil.
2048	4096	16384	65536	156250	312500	468750	625000

Zvyšné štyri body testov vypadajú následovne:

80 mil.	100 mil.	120 mil.	140 mil.
2,5 mil.	3,125 mil.	3,75 mil	4,375 mil.

Kedže testované algoritmy sú algoritmami vo vnútornej pamäti, je vhodné aby veľkosť

vstupných dát bola taká veľka, aby mohol algoritmus bežať čisto vo vnútornej pamäti bez nutnosti swapovania, čo je zrejme logické. Otázkou je, ako zistiť pre danú veľkosť vnútornej pamäte a veľkosť vstupných dát či daný proces pobeží bez nutnosti swapovania.

Dá sa to urobiť empiricky, nasledujúcim spôsobom. Systému sa zakáže swapovanie príkazom `swapoff -a` a postupne sa pre vstupné dáta, zväčšujúce sa po malých krokoch, otestujú naraz všetky metódy. V prípade že program spadne (skončí) je to výsledok toho, že v systému už nebola voľná pamäť – a tým dostaneme hornú hranicu veľkosti vstupných dát.

Aby sme zaručili, že merania testovaných algoritmov sú spoľahlivé, vykonali sme pre každý bod merania 30 opakovaní. Vrámci jedného opakovania jedného bodu, bežia všetky algoritmy nad tým istým vstupným poľom. Aby sme zaručili, že merania algoritmov nie sú vzájomne ovplyvnené tým, že sa v cache pamäti nachádzajú časti dát z behu predchádzajúceho algoritmu, implementovali sme funkciu na vyčistenie cache pamäte z [5]:

```
#define CBYTES (1<<19)
#define CINTS (CBYTES/sizeof(int))

static int dummy[CINTS];
volatile int sink;

void clear_cache()
{
	int i;
	int sum = 0;

	for (i = 0; i < CINTS; i++)
		dummy[i] = 3;
	for(i = 0; i < CINTS; i++)
		sum += dummy[i];
	sink = sum;
}
```

5.1.4 Testované algoritmy

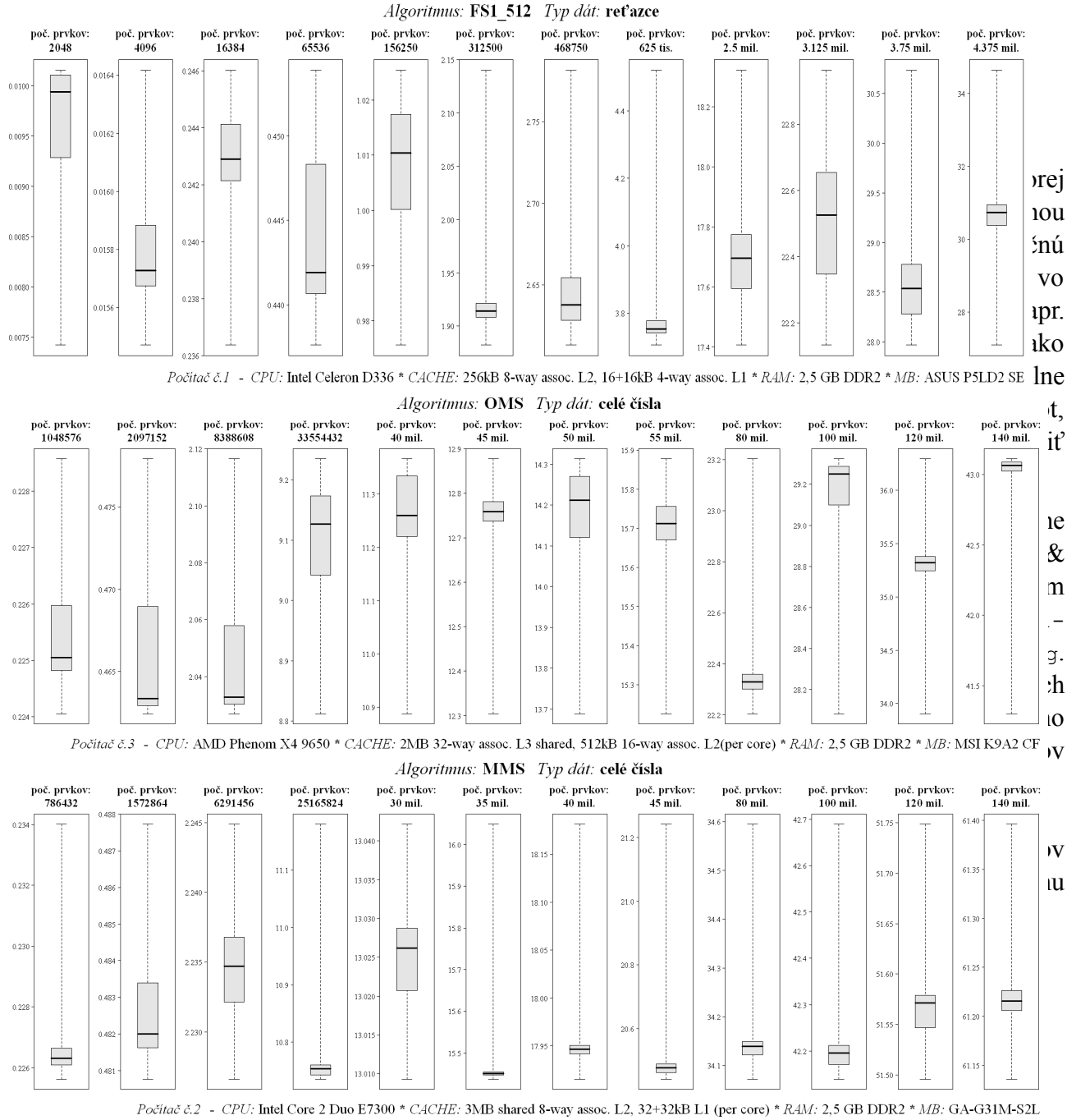
Pre realizáciu testov sme vybrali algoritmy popísané v [2] (Sekcia 4.1). Naimplementované boli všetky algoritmy okrem poslednej optimalizácie heapsortu a základného heapsortu. Tieto algoritmy sú zástupcami z triedy cache-aware algoritmov. Pre porovnanie sme naimplementovali aj zástupcu z triedy cache-oblivious algoritmov, lazy funnelsort.

Rozhodli sme sa neimplementovať generický lazy funnelsort, pretože veríme, že réžia spojená s vystavaním generického *k*-mergeru pre každú úroveň rekurzie by bola neúnosne veľká. Namiesto toho sme implementovali aproximáciu teoretického *k*-mergeru. To znamená, že algoritmus začína s pevne daným *k* a pri rekurzií používa *k*, ktoré je vždy mocninou dvojky. Ďalšia vec, ktorú sme oproti pôvodnému algoritmu upravili, je základný prípad rekurzie. Vytvorili sme dve varianty algoritmu. Prvá varianta algoritmu ukončí rekurziu v prípade keď hodnota *k* klesne pod číslo 4. Zostávajúce podpolia zotriedi základným quicksortom. Druhá varianta algoritmu ukončí rekurziu až v prípade, keď je počet prvkov podproblému menší ako 32. Zostávajúce podpolia zotriedi insertion sortom. Každá varianta lazy funnelsortu bola implementovaná pre pevné počiatočné *k* rovné 512 a 64. Výsledkom sú teda štyri algoritmy.

Dohromady sme implementovali dvanásť algoritmov:

BMS	obyčajný mergesort

OMS	základný mergesort zo Sekcie 4.1
TMS	tiled mergesort zo Sekcie 4.1
MMS	multimergesort zo Sekcie 4.1
BQS	základný quicksort zo sekcie 4.1
OQS	memory-tuned quicksort zo sekcie 4.1
MQS	multi-quicksort zo sekcie 4.1
DHS	d-árny heapsort zo sekcie 4.1
FS1_512	512-cestný lazy funnelsort, prvá varianta, zo Sekcie 4.2
FS1_64	64-cestný lazy funnelsort, prvá varianta, zo Sekcie 4.2
FS2_512	512-cestný lazy funnelsort, druhá varianta, zo Sekcie 4.2
FS2_64	64-cestný lazy funnelsort, druhá varianta, zo Sekcie 4.2

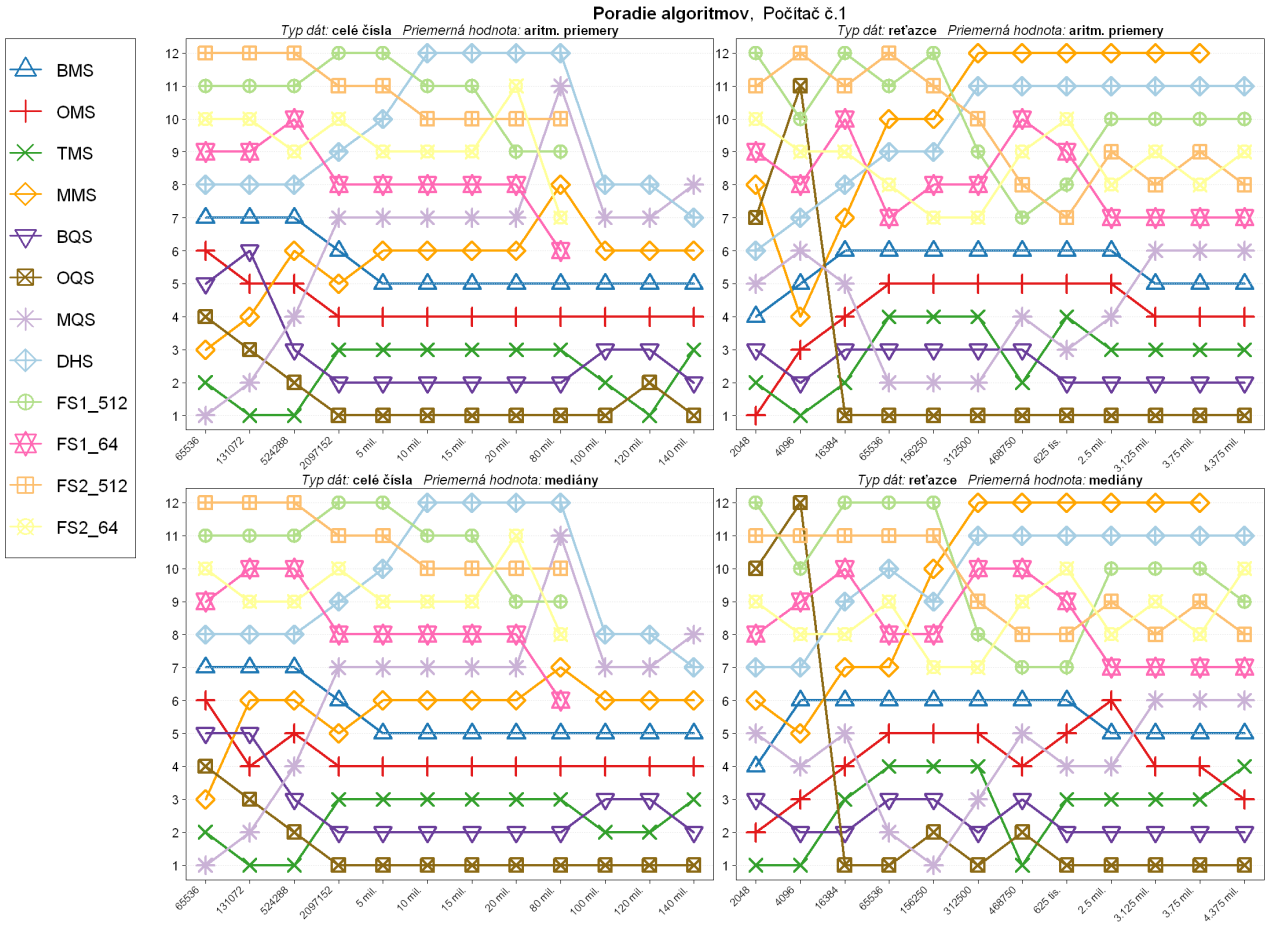


Obr. 5.1: Príklady Box & Whisker krabicových diagramov ukazujúce distribúciu nameraných dát. Zhora: algoritmus *FS1_512* triediaci reťazce na počítači č.1, algoritmus *OMS* triediace celé čísla na počítači č.3, algoritmus *MMS* triediaci celé čísla na počítači č.2.

týchto hodnôt by došlo k ich priblíženiu. Konkrétne sme vytvorili pre každý testovaný počítač tabuľky obsahujúce v stĺpcoch aritmetický priemer, medián, 1-useknutý priemer, 2-useknutý priemer, 3-useknutý priemer, rozdiel mediánu a aritmetického priemeru, rozdiel 1-useknutého priemeru a mediánu, rozdiel 2-useknutého priemeru a mediánu, rozdiel 3-useknutého priemeru a mediánu. Riadky reprezentovali merané body. Tieto tabuľky sa vo formáte csv nachádzajú na priloženom cdčku v adresári `'/vysledky/pocitac[1-4]/mmd'`. Je zvlášť súbor pre celé čísla a pre reťazce. Bohužial ani táto metóda nám jednoznačne nepovedala, akú priemernú hodnotu zvoliť. Rozdiely useknutých priemerov a mediánov vo väčšine prípadov nevykazovali monotónnu tendenciu (tzn. s rastúcim useknutím sa nezmenšovali).

Nakoniec sme sa rozhodli, že keď ani sledovanie rozdielov mediánu, aritmetického priemeru a α-useknutého priemeru nevykazovalo jasné výsledky, pozrieme sa na poradie algoritmov v prípade mediánu a aritmetického priemeru. Ak sa nebude výrazne líšiť poradie algoritmov pre

mediány a aritmetické priemery, zvolíme za priemernú hodnotu aritmetické priemery. Na priloženom cdčku sa v adresári `'/vysledky/pocitac[1-4]/poradie'` nachádzajú tabuľky vo formáte csv, obsahujúce dáta poradia algoritmov pre príslušný typ dát a priemernú hodnotu, a ďalej sa tam nachádzajú aj grafy odpovedajúce jednotlivým súborom s dátami. Formát dátových súborov a obrázkov je `'order_[mean | median]_[int | str]'`. Uvedieme jeden príklad pre počítač č.1:



Obr. 5.2: Poradie algoritmov na počítači č.1 pre priemernú hodnotu počítanú ako medián a aritmetický priemer. Vľavo hore – celé čísla, aritmetické priemery, vpravo hore – reťazce, aritmetické priemery, vľavo dole – celé čísla mediány, vľavo dole – reťazce mediány.

Ako vidíme z obrázku Obr. 5.2, poradie algoritmov sa výrazne nemení ak volíme za priemernú hodnotu mediány alebo artimetické priemery. Z toho dôvodu volíme za priemernú hodnotu aritmetické priemery.

5.3 Výsledok testov

Spracovaním všetkých našich testov sme dostali niečo vyše 200 rôznych tabuliek a grafov. Nie je v našich silách zahrnúť a okomentovať všetky grafy a preto uvádzame len výsledky analýzy našich

testov.

Malými poľami budeme nazývať prvé štyri body testovacích dát. Strednými poľami budeme nazývať stredné štyri body testovacích dát. Veľkými poľami budeme nazývať posledné štyri body testovacích dát.

Na počítači č.1 je pre malé polia a celé čísla najrýchlejší OQS a BQS. Medzi skupinu najrýchlejších algoritmov v tomto rozpätí ďalej patrí MMS, TMS, BMS, OMS. Pre stredné polia a celé čísla je najrýchlejší algoritmus OQS a BQS. Ďalej medzi skupinu najrýchlejších algoritmov patria TMS, OMS a BMS. Algoritmus MMS pre stredné polia a celé čísla zaznamenal výrazné zhoršenie. Pre veľké polia je najrýchlejším algoritmom OQS a BQS. Do skupiny najrýchlejších algoritmov ďalej patria TMS, OMS a BMS. Pre malé polia a reťazce je na počítači č.1 najrýchlejší OQS, BQS a MQS. Ďalej medzi najrýchlejšie algoritmy v tomto rozpätí patria TMS, OMS a BMS. Pre stredné polia a reťazce je najrýchlejším algoritmom OQS, BQS a MQS. Medzi najrýchlejšie algoritmy v tomto rozpätí ďalej patria TMS, OMS a BMS. Pre veľké polia a reťazce je najrýchlejším algoritmom OQS, BQS. Medzi najrýchlejšie algoritmy ďalej patria TMS, OMS a BMS. Algoritmus MQS zaznamenal pre veľké polia a reťazce výrazné zhoršenie.

Na počítači č.2 je pre celé čísla a malé polia najrýchlejší algoritmus OQS a BQS. Ďalej medzi najrýchlejšie algoritmy patrí TMS, OMS a BMS. Pre stredné polia a celé čísla je výrazne najrýchlejší OQS a BQS. Ďalej medzi najrýchlejšie algoritmy patria TMS, OMS a BMS. Pre veľké polia a celé čísla je výrazne najrýchlejší znovu OQS a BQS. Ďalej medzi najrýchlejšie algoritmy patria TMS, OMS a BMS. Pre malé polia a reťazce je najrýchlejší MQS. Ďalej medzi najrýchlejšie algoritmy v tomto rozpätí patria OQS, BQS a TMS. Pre stredné polia a reťazce je najrýchlejší algoritmus znovu MQS. Ďalej medzi najrýchlejšie algoritmy patria OQS, BQS a TMS. Pre veľké polia a reťazce je skupina najrýchlejších algoritmov MQS, OQS, BQS a TMS. V tomto rozpätí sú medzi najrýchlejšími algoritmami rozdiely veľmi malé.

Na počítači č.3 je pre celé čísla a malé polia najrýchlejší algoritmus TMS. Medzi skupinu najrýchlejších algoritmov patrí OMS, BMS, OQS a BQS. Pre stredné polia a celé čísla je najrýchlejším algoritmom znovu TMS. Medzi skupinu najrýchlejších algoritmov ďalej patria OQS, BQS, OMS a BMS. Pre celé čísla a veľké polia sú pre prvý bod najrýchlejšie algoritmy TMS a OMS, ale v treťom a štvrtom bode (teda pre veľkosti polí 1ľ0 mil. a 140 mil.) sú najrýchlejšie algoritmy OQS a BQS. Medzi najrýchlejšie algoritmy v tomto rozpätí patrí ešte BMS. Pre malé polia a reťazce sú najrýchlejšie algoritmy TMS, OMS a MQS. Ďalej medzi najrýchlejšie algoritmy v tomto rozpätí patria OQS, BQS, MQS. Pre stredné polia a reťazce je najrýchlejší TMS. V prvých dvoch bodoch bol MQS rýchlejsí jako OMS, ale v poslednom bode sa to vymenilo. Medzi skupinu najrýchlejších algoritmov v tomto rozpätí patria OQS, BQS a BMS. Pre veľké polia a reťazce bol najrýchlejší algoritmus TMS a OMS. Ďalej medzi skupinu najrýchlejších algoritmov v tomto rozpätí patria OQS, BQS, BMS a MQS.

Na počítači č.4 sú pre celé čísla a malé polia najrýchlejšie algoritmy TMS, OQS, BQS a OMS. Medzi skupinu najrýchlejších algoritmov v tomto rozpätí ešte patrí BMS. Pre stredné polia a celé čísla sú najrýchlejšie algoritmy TMS a OMS. Medzi skupinu najrýchlejších algoritmov ďalej patria OQS, BQS a BMS. Pre veľké polia sú pre prvé tri body najrýchlejšie algoritmy TMS a OMS, ale pre posledný bod sú najrýchlejšie OQS a BQS. Medzi skupinu najrýchlejších algoritmov ďalej patrí ešte BMS. Pre malé polia a reťazce sú najrýchlejšie algoritmy MQS a TMS. Medzi skupinu najrýchlejších algoritmov v tomto rozpätí ďalej patria OQS, BQS, OMS a BMS. Pre stredné polia a reťazce sú najrýchlejšie algoritmy znovu MQS a TMS. Medzi skupinu najrýchlejších algoritmov

ďalej patria OMS, OQS, BQS a BMS. Pre veľké polia a reťazce je najrýchlejším algoritmom TMS. Medzi skupinu najrýchlejších algoritmov ďalej patria OMS, OQS a BQS. Algoritmus MQS v tomto rozpätí zaznamenal výrazné zhoršenie.

Z výsledokv testov vyplýva, že ak aj na určitých miestach bežali optimalizované algoritmy v priemere rýchlejšie ako obyčajné algoritmy, tento rozdiel bol veľmi malý. Na niektorých miestach, zvlášt pre najväčšie veľkosti vstupných dát, boli v priemere najrýchlejšie obyčajné algoritmy alebo ich základné varianty s nepamäťovými optimalizáciami. Predkladám preto k úvahe, či má zmysel zbytočne optimalizovať algoritmus, ak vo výsledku je čas behu daného algoritmu pomalší ako obyčajný algoritmus, z ktorého sme vychádzali. Dôvodom, prečo optimalizované cache-aware varianty a cache-oblivious algoritmy nevykazovali očakávaný nárast v rýchlosti, môže byť neznámy konštantný faktor medzi asymptoticky optimálnou (cache) zložitosťou algoritmu v ideal-cache alebo I/O modeli a jeho reálnou (cache) zložitosťou v reálnom pamäťovom systéme. Je zrejmé, že pre skúmané implementácie algoritmov na skúmaných počítačoch, očakávaná úspora znížením predpokladaného počtu cache výpadkov nevyrovnala stratu spôsobenú nárastom komplikovanosti algoritmu. Z našich výsledkov vyplýva, že ani na moderných počítačoch s niekoľkými úrovňami cache pamäte použitím obyčajného alebo mierne optimalizovaného obyčajneho algoritmu nič nepokazíme.

Kapitola 6

Záver

Cieľom našej práce bolo predovšetkým prakticky overiť moderné prístupy k problému triedenia dát na moderných počítačoch s cache pamäťou. Nižšie uvádzame prehľad našej práce:

V Kapitole 2 sme nahliadli do spôsobu akým pracujú moderné pamäťové systémy a videli sme jako pamäťová latencia a vzory prístupov do pamäte algoritmu ovplyvňujú čas jeho behu. Pomocou príkladu sme ukázali, že dopad latencie pamäte na čas behu algoritmov nie je možné zistiť analýzou v modeli RAM.

Ideal-cache model a ďalšie modely pamäťových systémov vedia zachytiť dopad efektov a latencie pamäťového systému na skúmaný algoritmus. Pri skúmaní týchto modelov v Kapitole 3 sme videli , že ideal-cache model poskytuje oproti ostatným modelom určité výhody. Pomocou overenia predpokladov sme ukázali, že ideal-cache model je možné zredukovať tak aby odpovedal reálnym pamäťovým systémom a že algoritmus, ktorý je asymptoticky optimálny v ideal-cache modeli, je tiež asymptoticky optimálny v reálnych pamäťových systémoch.

Pre potreby praktického overenia moderných prístupov k problému triedenia dát sme potrebovali zástupcov dvoch hlavných prístupov k problematike. Preto v Kapitole 4 popisujeme algoritmy, ktoré zastupujú triedu cache-aware a cache-oblivious algoritmov.

V Kapitole 5 sme popísali spôsob akým sme vykonali testy a akú hypotézu sme testami vlastne sledovali. Myslíme si, že testy boli vykonané korektným spôsobom a na dostatotočnom počte rôznych počítačov dobre reprezentujúcich moderné pamäťové systémy.

Z výsledkov našich testov vyplýva, že pre obmedzenú veľkosť vnútornej pamäte nemusí vždy platiť, že čím lepšie teoretické výsledky algoritmus dosahuje, tým bude rýchlejší. V prostredí našich testov mali najlepšie výsledky obyčajné a mierne optimalizované algoritmy, kde tieto optimalizácie neboli pamäťové.

Dodatok A
Štruktúra priloženého média

Nasleduje stručným popis obsahu priloženého média:
`/src/ukazka` – program zo Sekcie 2.3
`/src/test` – zdrojové súbory algoritmov a testov
`/src/r` – zdrojový súbor štatistického programu R
`/vysledky/pocitac[1-4]/` – namerané výsledky našich testov
- `box_whisker/` – Box & Whisker krabicové diagramy pre daný počítač
- `mmd/` – csv súbory obsahujúce tabuľky rozdielov mediámu, aritm. priemeru a L-hodnot
- `poradie/` – csv súbory a grafy poradia algoritmov pre daný počítač
- `result_comps/` – csv súbory a grafy poradia algoritmov pre počet porovnaní pre daný počítač
- `result_copys/` – csv súbory a grafy poradia algoritmov pre počet kopírovaní pre daný počítač
- `result_time/` – csv súbory a grafy času behov algoritmov

Použitá Literatura

[1] Aggarwal A., Vitter J. S. (1988): The input/output complexity of sorting and related problems . *Communications of the ACM* 31(9) , 1116-1127.

[2] Ladner E.R., LaMarca A. (1998): The Influence of Caches on the Performance of Sorting. *Journal of Algorithms* 31, 66-104.

[3] Sedgewick R.: Implementing quicksort programs. *Communications of the ACM 21*(10), 847-857.

[4] Frigo M., Leiserson C.E., Prokop H., Ramachandran S. (1999): Cache-Oblivious Algorithms. *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 285-297.

[5] Bryant R.E., Ohallaron D.R. (2003): Computer Systems, A Proggramer's Perspective. (2001, manuscript) Prentice Hall.

[6] Patterson D.A., Hennessy J.L. (1996): Computer Organization and Design: The Hardware/Software Interface, 2nd ed., Morgan Kaufmann Publishers Inc.

[7] Knuth D.E. (1973): The Art of Computer Programming, Vol. 3. - Sorting and Searching, Addison-Wesley, Reading, MA.

[8] Johnson D.B. (1975): Priority queues with update and finding minimum spanning trees. *Inform. Process. Lett. 4*.

[9] Wolfe M. (1989): More iteration space tiling. *Procceedings of Supercomputing '89, 655-664.*

[10] LaMarca A., Ladner E.R. (1996): The influence of Caches on the performance of heaps. *Journal of Experimental Algorithmics 1(4)*.

[11] Brodal G. S., Fagerberg R. (2002): Cache Oblivious Distribution Sweeping. *Procceedings of the 29th International Colloquium on Automata, Languages and Programming*,*426-438.*

[12] Cormen T.H., Leiserson C.E., Rivest R.L. (1990): Introduction to Algorithms. MIT Press and McGraw Hill, 189.

[13] A.Aggarwal, B.Alpern, A.K.Chandra, M. Snir, A model for hierarchical memory, Proceedings of the 19th Annual ACM Symposium on Theory of Computing, ACM Press (1987), 305-314.

[14] Aggarwal A., Chandra A.K., Snir M. (1987): Hierarchical memory with block transfer. *Proceedings of the 28th Annual IEEE Symposium on foundations of Computing*, IEEE Computer Society Press, 204-216.

[15] Srivastava A., Eustace A. (1994): ATOM A system for building customized program analysis tools. *Proceedings of the 1994 ACM Syposium on Programming Languages Design and Implementation*, Assoc. Comput. Mach., New York, 194-205.

[16] Hennessy J.L., Patterson D.A. (1996): Computer Architecture: A Quantitative Approach, 2nd edition. Morgan Kaufman, San Mateo, CA.

[17] Töpfer P. (1995): Algoritmy a programovací techniky, Prometheus.

[18] Huang B.C., Langston M.A. (1988): Practical in-place merging. *Communications of the ACM,* 31(3).

[19] Lowney P., Freudenberger S., Karzes T., Lichtenstein W., Nix R., Odonnel J., Ruttenberg J. (1993): The multiflow trace scheduling comiler. *Journal of Supercomputing*, 7:51-142.

[20] Robert W. Floyd. Algorithm 245 – Treesort 3, 1964, CACM 7(12): 701.

[21] Vitter J.S. (2001): External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys* 33, 2, 209-271.

[22] Sleator D.D., Tarjan R.E.: Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM* 28,2,202-208.

[23] Motwani R., Raghavan P.: Randomized Algorithms. Camridge Univerzity Press.

[24] Prokop H.: Cache-Oblivious Algorithms, M. Sc. Thesis. Massachusetts Institute of Technology.

[25] Brodal G.S., Fagerberg R. (2003): On the limits of cache-obliviousness. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*.

[26] Bojesen J., Katajainen J., Spork M.: Performance engineering case study: heap construction. *Journal of Experimental Algorithmics 5*.

[27] Drepper U. (2007): What every programmer should know about memory. Red Hat Inc..

[28] Demaine E. (2001): Cache-Oblivious Algorithms and Data Structures. *EFF Summer School on Massive Data Sets*.

[29] Brodal G.S., Fagerberg R., Vinther K. (2008): Engineering a Cache-Oblivious Sorting Algorithm. *ACM Journal of Experimental Algorithmics*, Vol. 12, 2.2.